ARCHITECTURE FOR DETECTING INFINITE LOOPS OF WEB SERVICES USING TIME BOUNDARY VALUES

Miss Nattapatch Srirajun

A Dissertation Submitted in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy Program in Computer Science and
Information Technology
Department of Mathematics and Computer Science
Faculty of Science
Chulalongkorn University
Academic Year 2013

สถาปัตยกรรมเพื่อการตรวจหาวงวนไม่รู้จบของเว็บเซอร์วิสโดยใช้ค่าขอบเวลา

นางสาวณัฐพัชญ์ ศรีราจันทร์

วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิทยาศาสตรดุษฎีบัณฑิต
สาขาวิชาวิทยาการคอมพิวเตอร์และเทคโนโลยีสารสนเทศ ภาควิชาคณิตศาสตร์และวิทยาการคอมพิวเตอร์
คณะวิทยาศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย
ปีการศึกษา 2556

Thesis Title	ARCHITECTURE FOR DETECTING INFINITE LOOPS OF WEB SERVICES USING TIME BOUNDARY VALUES
By	Miss Nattapatch Srirajun
Field of Study	Computer Science and Information Technology
Thesis Advisor	Assistant Professor Pattarasinee Bhattarakosol, Ph.D.
Thesis Co-Advisor	Associate Professor Panjai Tantasanawong, Ph.D.

Accepted by the Faculty of Science, Chulalongkorn University in Partial Fulfillment of the Requirements for the Doctoral Degree

....Dean of the Faculty of Science

(Professor Supot Hannongbua, Dr.rer.nat.)

THESIS COMMITTEE

....Chairman

(Assistant Professor Rajalida Lipikorn, Ph.D.)

....Thesis Advisor

(Assistant Professor Pattarasinee Bhattarakosol, Ph.D.)

....Thesis Co-Advisor

(Associate Professor Panjai Tantasanawong, Ph.D.)

....Examiner

(Arthorn Luangsodsai, Ph.D.)

....Examiner

(Associate Professor Taratip Suwannasart, Ph.D.)

....External Examiner

(Assistant Professor Benchaphon Limthanmaphon, Ph.D.)

ณัฐพัชญ์ ศรีราจันทร์ : สถาปัตยกรรมเพื่อการตรวจหาวงวนไม่รู้จบของเว็บเซอร์วิสโดยใช้ค่าขอบเวลา. (ARCHITECTURE FOR DETECTING INFINITE LOOPS OF WEB SERVICES USING TIME BOUNDARY VALUES) อ.ที่ปรึกษาวิทยานิพนธ์หลัก: ผศ. ดร.ภัทรสินี ภัทรโกศล, อ.ที่ปรึกษาวิทยานิพนธ์ร่วม: รศ. ดร.ปานใจ ธารทัศนวงศ์, 79 หน้า.

เว็บเซอร์วิสเป็นระบบระยะไกลที่ถูกพัฒนาขึ้นบนระบบแบบกระจายที่มีผู้ร้องขอและผู้ตอบสนอง เพื่อการรับประกันการบริการที่ดีนั้น ผู้พัฒนาซอฟท์แวร์จำเป็นต้องหาเทคนิคต่าง ๆ เพื่อการตรวจจับและป้องกันความผิดพลาดในขณะการทำงาน ปัญหาหนึ่งที่มีความสำคัญอย่างยิ่งของการบริการคือ สภาวะการวนไม่รู้จบ เนื่องจากระบบการทำงานจะไม่ตอบสนองผลลัพธ์ใดๆ ให้แก่ผู้ร้องขอการบริการ นอกจากนี้แล้วทรัพยากรของระบบจะมีการใช้งานอย่างสูงจนระบบไม่สามารถทำงานได้อีกต่อไป ดังนั้นงานวิจัยนี้ได้นำเสนอสถาปัตยกรรมที่ทำงานบนเว็บเซอร์วิส อันเกี่ยวข้องกับอัลกอริทึมที่ทำงานชนิดไม่สามารถกำหนดการวนได้ และเพื่อแก้ปัญหาการวนไม่รู้จบนั้น งานวิจัยนี้ได้นำเสนอเทคนิคเพื่อการแก้ปัญหาสองวิธี คือ การตรวจสอบเวลาการทำงานด้วยการกำหนดขอบของระยะเวลาการทำงาน และการตรวจสอบรูปแบบของค่าตัวแปรที่ใช้วน เทคนิคนี้สามารถตรวจจับและควบคุมสภาวะการเกิดการวนไม่รู้จบ ก่อนที่ความเสียหายจะเกิดขึ้นกับระบบการทำงาน ผลการทดลองที่เกิดจากการใช้สถาปัตยกรรมที่นำเสนอนี้จะเป็นการวัดประสิทธิภาพการใช้หน่วยความจำหลัก ซึ่งผลที่ปรากฏได้แสดงให้เห็นว่า สถาปัตยกรรมที่นำเสนอนี้สามารถเพิ่มประสิทธิภาพการใช้หน่วยความจำหลักได้ นอกจากนี้แล้ว สภาวะการวนไม่รู้จบสามารถถูกตรวจจับและยกเลิกได้ด้วยกลไกการทำงานของสถาปัตยกรรมนี้

ภาควิชา	คณิตศาสตร์และวิทยาการคอมพิวเตอร์	ลายมือชื่อนิสิต ....
		ลายมือชื่อ อ.ที่ปรึกษาวิทยานิพนธ์หลัก ....
สาขาวิชา	วิทยาการคอมพิวเตอร์และเทคโนโลยีสารสนเทศ	ลายมือชื่อ อ.ที่ปรึกษาวิทยานิพนธ์ร่วม ....
ปีการศึกษา	2556	

5173821423 : MAJOR COMPUTER SCIENCE AND INFORMATION TECHNOLOGY
KEYWORDS: EXECUTION TIME / MEAN TIME INTERVAL / WEB SERVICES / INFINITE LOOP

NATTAPATCH SRIRAJUN: ARCHITECTURE FOR DETECTING INFINITE LOOPS OF WEB SERVICES USING TIME BOUNDARY VALUES. ADVISOR: ASST. PROF. PATTARASINEE BHATTARAKOSOL, Ph.D., CO-ADVISOR: ASSOC. PROF. PANJAI TANTASANAWONG, Ph.D., 79 pp.

A web service is a remote system, implemented on a distributed system with request and response. To guarantee good services, software developers must find techniques for detecting and preventing errors during run-time. One problem that is important to such services is the infinite-loop situation, since the system will not return any results to the requester. Moreover, the resources of the system are overloaded until the server cannot function. Therefore, this research proposes the web service architecture relating to non-deterministic loop algorithms. For solving the infinite-loop problem, this research proposes two techniques: checking execution time by defining the execution time boundary, and checking patterns of iteration variable values. The techniques can detect and control the infinite-loop situation before causing damage to the system. The experiments using the proposed architecture measured the CPU usage. The results show that the architecture can improve the CPU usage. Moreover, the infinite-loop situation can be detected and terminated by the mechanisms of this architecture.

Department:	Mathematics and Computer Science	Student's Signature ....
Field of Study:	Computer Science and Information Technology	Advisor's Signature ....
Academic Year:	2013	Co-Advisor's Signature ....

ACKNOWLEDGEMENTS

During studying for a Ph.D., one of completion is to look over the past journey and remember all the friends and family who have helped and supported me along this long way. I would like to thank the Office of Higher Education Commission and Chulalongkorn University for their financial support.

I would like to express my heartfelt gratitude to Assist.Prof. Dr.Pattarasineee Bhattarakosol, Assoc.Prof. Dr.Panjai Tantasanawong, and Prof. Dr.Sunyoung Han, for their advice, guidance and care, which help me overcome all difficulties of the process of research and make this dissertation possible.

I would also like to thank dissertation committee, Assist.Prof. Dr.Rajalida Lipikorn, Dr.Arthorn Luangsodsai, Assoc.Prof. Dr.Taratip Suwannasart, and Assist.Prof. Dr.Benchaphon Limthanmaphon, who provided encouraging and constructive feedback. It is no easy task, reviewing a thesis, and I am grateful for their thoughtful and detailed comments.

Finally, I would also like to thank all lecturers and colleagues at Innovative Network and Software Engineering Technology (INSET) laboratory, the Department of Mathematics and Computer Science, Faculty of Science, Chulalongkorn University, for their warmest care and support.

CONTENTS

Page

Page

Page

LIST OF TABLE

Page

LIST OF FIGURE

Page

CHAPTER 1
INTRODUCTION

1.1 Introduction and Problem Review

Currently, various software developers use web services to provide and share services among similar tasks of different organizations. The implemented web service is stored in a web server. Moreover, one server can have many web-service methods. Each web-service method will perform each task requirement. The web- service method structure is similar to that of general software functions. Therefore, the standard for performing a task can be settled.

The web-service method structure with Microsoft Visual C# language was shown in Figure 1.1.

```
[WebMethod]
Web-service method name (input parameters)
{
        <statement lists>
        return values;
}
```

Figure 1.1 The structure of web-service method

The web service development process is similar to software development process since web services can be counted as a part of the developed software over the network. Therefore, the quality control mechanism during the software process is also applied while web services are implemented. As a result, in order to remove defects from the developed web agents during software process, two processes must be performed: verification and validation processes.

The verification process attempts to detect all functional defects before the software is delivered, while the validation process attempts to ensure that the developed web service can serve user's expectations. Nevertheless, some errors can occur after the software has been delivered to users, especially run-time errors. Run-time errors are a significant problem that is hard to detect during the testing period since the test data set might not cover all possible inputs from users. Thus, every web agent cannot be fully guaranteed to be error-free.

Once run-time errors occur, the reliability of the software drops [1]. Moreover, the software overhead to re-perform the task is increased. Based on the run-time problems mentioned, the situation of web services is similar to other programs in that some critical processes can exist. Since there are various critical errors during run-time processes, this research will focus on non-responsive services under the infinite-loop situation in which there is no-response to clients. The proposed mechanism tends to detect and protect from any damages that can arise when infinite loops occur. The following section elaborates the focused domain of this research.

1.2 Problem Domain: Non-Responsive Web Services under the Infinite-Loop Situation

As mentioned previously, non-responsive web services under the infinite-loop situation is a type of critical run-time errors. Thus, critical damage can occur and the CPU resource usage is high but unused. Consequently, the Quality of Service (QoS) of web services must be considered and maintained.

Based on the proposed QoS model by Araban and Sterling [2], QoS factors for reliability of web services depending on the correctness of the execution services. Moreover, the performance of the services is measured from the space usage efficiency during execution and the execution time for each round. Furthermore, the usability of the web services is based on the types and values of the input and output parameters.

Thus, the problem of non-responsive web services under the infinite-loop situation can be considered as the reliability of the services that affects the performance of the system, since space usage during run-time is large without benefits. Additionally, the value of input and output parameters that relate to the control loop condition must be determined.

Therefore, this research will focus on the non-responsive web services under the infinite-loop situation since its impact is critical to all the important factors of QoS.

1.3 Research Objectives

The objectives of this research are as follows:

1. To design a mechanism for detecting and controlling the infinite-loop situations,
2. To design the architecture of distributed web services that guarantees dynamic-loop process.

1.4 Expected Outcomes

The expected outcomes of this research are as follows:

1. The mechanism that allow the infinite-loop problem of the service process to be controlled and notified to the requester;
2. Architecture that can guarantee the performance and reliability of web-services agent that contains dynamic loops;
3. The architecture that can perform real time verification.

1.5 Scopes of the Study

1. Web services architectures studied under the SOA (Service Oriented Architecture) environment of service providers.
2. The research focuses on at least four computing intensive web services contain dynamic loops.
3. Web services being tested cover five patterns: sequence, parallel, choice, loop, and compound compositions.
4. The termination condition of the loop is dependent on input parameters only; and the termination condition is in the computing model.
5. The value of the detecting infinite-loop situation will be dependent on the upper bound of the EMTI (Execution Mean Time Interval).
6. The EMTI value will be adjusted by the learning data set obtained from users' execution time.
7. The network environment under the composite web services test has an effect on the EMTI value.
8. The implementation is developed using the Microsoft Visual C# platform.

1.6 The Definition of Terms

Deterministic loop: A deterministic loop is a predictable loop. The number of iterations of a loop is known before the loop has started.

Non-deterministic loop: A non-deterministic loop is not a predictable loop. A loop is driven by outside factors such as user input. Therefore, the number of iterations is not known in advance before the loop has started.

Static loop: the static loop has a constant number of loop cycles in every loop execution.

Dynamic loop: the dynamic loop has a variable number of the loop cycles. The number of loop cycles is not constant, depending on the external factors such as called function, input value from the user, etc.

Iteration variable [3]: the iteration variable is a variable at the loop condition. It is a factor for controlling the loop cycle.

Critical section: a situation of a loop that has a very low probability of exiting according to the control loop condition; or the probability of exiting the loop is close to zero.

Non-critical section: a situation of a loop that has a high probability of exiting according to the control loop’s condition; or the probability of exiting the loop is close to one.

CHAPTER 2
THEORIES AND LITERATURE REVIEWS

In this chapter, related articles are reviewed under the problem of the non-responsive web services with the infinite-loop situation. First, Section 2.1, the software at run-time situation will be presented which relates to the loop instruction and the infinite-loop situation. Second, Section 2.2, the packet time-out mechanism technique will be presented for consideration for checking the execution time. Moreover, in Section 2.3, the confidence of execution time boundary will be proposed, which identifies the execution mean time interval. Finally, the literature reviews of web services are drawn in Section 2.4. Details of each section will be described as follows.

2.1 Software at Run-time Situation

It is the fact that the success of every organization around the world relies on the quality of implemented software. Moreover, software reliability is also an important issue that the developers must consider. The reliability of software is dependent on the number of errors that can be detected and prevented during the software processes. In order to obtain qualified software, there are many factors that have to be considered during the software development process. In addition, the structure of software design relating to the implemented parameters is another protection technique that has to be considered.

Based on the traditional development process, the input data set is an important factor that leads to software implementation. Therefore, the simulation models [4-5] were proposed in the verification and validation phases. These proposed models increase the confidence of software usage during run-time. However, the complexities of these models rely on the input data set generator.

Since the run-time error problem is a significant issue for the success or failure of software usages, it has been focused on by many researchers. The event of changing states in the system and the execution time are also considered as other factors for discovering the errors hidden in the software. The design method for testing running times of objects was proposed under the object-oriented environment[6]. The method, called as the real time logic (RTL), was applied in the testing state and was a constraint for the changing software state. The result showed that the model can detect the constraint violation. Unfortunately, this run-time checker used a static data-base for checking; therefore, its value cannot be altered in the run-time environment.

In addition, reliability can be obtained by adding a debugging module [7]. This research proposed result-checking, simple checkers, self-correctors on the possible input data, and time boundary when the software was executed. Moreover, a proposed monitoring and checking framework [8], called the Monitoring and Checking (MAC) architecture, used a run-time checker mechanism to fill the gap between the static verification and the testing mechanism of the software development process. Furthermore, Anomaly Detection by Resource Monitoring (Ayaka) [9] monitored and detected anomalies using only some system resource usage information. They also used a completely black-box approach based on machine learning methods to find anomalies, by comparing the application resource usage with the learned model.

The timing technique has also been applied to monitor the real-time system using the graph diagram [10]. This diagram is created to detect the violation during the event transfer period. This was implemented using Java run-time timing constraint monitor (JRTM). This method provided the shortest latency and low overhead for violation detection. In 2008, a method was proposed using the automata theorem for analyzing the time of program events in run-time monitoring [11]. The expected outcome is error detection. This prototype has revealed itself to be efficient with respect to real-time operating system deployment and decreases event overhead for increasing system performance.

Since the reliability of software is a significant consideration that is mostly determined during run-time process, therefore, the problems that can occur to obstruct the reliability issue can be basically protected using the exception class in software. Unfortunately, this technique cannot protect all kinds of the unwanted cases such as the long run-time problem.

The research in [12] proposed the theoretical approval to analyze a loop during run-time on primitive recursive functions, bounding the running time and complexity classes. This proposed approval can be applied to real systems. The experiment was verified by formal verification method. The result showed that result-checking may improve debugging tools for reliable systems.

Moreover, the Dynamic Invariant Detection U Checking Engine (DIDUCE) was proposed as a tool for detecting complex-program faults in the debugging process

[13]. This tool focuses on various faults which are failure from some inputs, failures in long-running programs, in component-based software. Additionally, the programs are tested with inputs for which the correct outputs are unknown. As a consequence of using four applications, this method can discover different types of faults, such as faults from algorithms, inputs, and developer’s misconceptions of the APIs.

According to the previous paragraphs, it is obvious that the processing time is of importance for the performance of the real-time system. Especially, the long-time running should be concerned. Thus, loop instructions and the infinite-loop situation will be described in the following sections.

2.1.1 Loop Instructions

A loop instruction may cause an infinite cycle depending on the condition of the loop instruction. It affects the resources of the running system, which are used enormously, such as memory and CPU usage. Incorrectness of software implementation has significant effects for critical software, such as finance software, shipping software, medical software, and scientific software, etc.

Loop characteristics have been defined in [14] which can be classified in three categories: a static-control loop, a conditional control loop, and a variable-dependent loop. The first category is the static-control loop, called as a well-structured loop. This type of the loop performs in static execution time, such as *for (i= 1; i< 10; i++) {…};.*

The second category is the conditional control loop, such as *while (1), do {…if (i<j), break, …};.* The execution of the loop will be terminated when the condition in the loop is satisfied. Therefore, the execution time is predictable based on its execution profile. The category is defined as an ill-structured loop.

The last category of loop is the most significant loop since the termination of the loop is dependent on the value of a variable at run-time, such as *while (i<j) do{…};.* Thus, this category is called a variable-dependent loop. Moreover, the countable loop [3] was proposed on characteristics, definition and restriction of the loop.

2.1.2 Infinite-Loop Situation

Referring to the defined loop characteristics above, there are two groups of the loops, which can be classified as: the deterministic loop, and the non-deterministic loop. The deterministic loop has the characteristics of the static control loop, while the non-deterministic loop is dependent on the conditional control loop and the variable-dependent loop because the termination of the loop occurs during the run-time process only. Consequently, the critical effect relies on the non-deterministic loop since users cannot predict the computation outcomes during the execution period. Furthermore, the infinite-loop situation can occur.

The infinite-loop problem is a situation when the system cannot release the system's resources; the performance of the system will be reduced. Therefore, the infinite-loop situation can occur only on non-deterministic loop algorithms, where there is no termination of non-deterministic loop. The infinite loop is defined in meaning as: "An infinite loop is an instruction sequence that loops endlessly when a terminating condition has not been set, cannot occur, and/or causes the loop to restart before it ends." [15].

Research relating to loop verification and detection such as loop checking was analyzed for the logic program in [16]. A new complete-loop checking mechanism with the key technique to expanded variants was developed, called VAF-checks (variant atoms loop checks for logic programs with functions). The key structural characteristics of infinite loops were captured. In addition, the LOOPER technique was proposed in [17] with an automated technique for dynamically analyzing a running program to design non-terminating programs. Symbolic execution in argument values was analyzed with a construct of satisfiability modulo theories (SMT). Moreover, studying the design model for developing dynamic software was recommended in [18], with a self-adaptive system. This system was suggested in parts of development methods. Therefore, the proposed technique with tools showed the dynamic self-adaptive behavior and the loop control.

Moreover, the Jolt system [19] presented dynamically detecting and escaping infinite loops. This system recorded the state change at the start of the loop instruction. If the previous and the current loop iterations produce the same state, this system will report to the user that the application is in an infinite loop. The system was designed to add instructions into the loop body. This system escaped loops by adding instruction into the loop body whenever the infinite loop occurred.

Size-change termination was proposed in [20]. This research considered on parameter-size analysis with graphs and proved that the program flow was recognized as causing infinite descent. In addition, the new program termination prover[21] was proposed using path-sensitive and context-sensitive program analysis. A tool was designed for checking the balance between the constructing and termination arguments. Moreover, the research in [22] presented an automatic non-termination checking with symbolic testing, using an automated generation of invariants which showed unreachable from the terminating states of a program.

The program termination is an invariant of the path, or well-found path. Binary reachability was used for finding a rank-function synthesis. On the other hand, partial evaluation was used for binding-time analysis for guaranteeing the termination of specialized program. Therefore, execution-time analysis for program termination should be considered, such as the program at risk of an infinite-loop situation.

The research about loop invariants was designed with a formula with mechanically-generated inductive assertions [23]. The mechanisms consist of generating all paths corresponding to every basic cycle of the loop, test conditional statements and loops, and generate a constraint on the parameters with a quantifier-free formula. The conjunction of all constraints on parameters is evaluated based on satisfying the loop invariant. From Figure 2.1, the while-loop has two testing condition paths in the loop for analyzing the loop invariant.

```
product (X, Y : integer) returns z: integer
var x,y: integer end var
<x, y, z>:=<X, Y, 0>;
while y ≠ 0 do
    if y mod 2 = 1    then <x, y, z>:=<2x, (y – 1) div 2, x + z>;
    else y mod 2 = 0 then <x, y, z>:=<2x, y div 2, z>;
    end if
end while
```

Figure 2.1 An example of simple calculation program using two inputs [23]

2.2 Packet Time-out Mechanism

Similarly to software, every packet in communication channels must have time to live (TTL) after being delivered in order to prevent a congestion situation in the communication line. Thus, when a packet is delivered and flowed into the communication channel, the packet will be terminated whenever it reached the TTL value; the sender will be informed by ICMP [24] to retransmit the data.

TTL represents the lifetime of any packet over the network. The value is set from zero to 255 and decremented by one when the packet passes each router across the network. If the TTL value reaches zero, the packet will be discarded. Thus, this mechanism can prevent congestion when a packet cannot reach the destination.

In this research, the packet time-out is adapted to find a software process lifetime. A boundary time was estimated for every execution using the concept of identifying Execution Mean Time Interval (EMTI) of the execution time.

2.3 Identifying Execution Mean Time Interval (EMTI)

The confidence interval [25] was used for creating a trusted execution time of the architecture. Normally, the execution time of a process is measured in the unit of "millisecond" (ms) or "second" (sec). For example, the boundary of the execution time of process A is not over 2500 ms. So, when the process is repeatedly executed, there will be an Execution Mean Time (EMT) obtained from the average value of every execution time of the process in a certain period.

The EMTI can be calculated based on the assumption that the distribution of the execution time is normal. The *EMTI* is calculated from the Execution Mean Time (*EMT*) of the service execution times. The formula for the EMTI is based on the confidence interval, as shown in the equation below.

$$\mathrm{EMTI} = \bar{x} \pm z_{\frac{\alpha}{2}} \frac{s}{\sqrt{n}}$$

where $\bar{x}$ is the EMT during a certain period,

α is the significance level,

Z is a distribution with confidence level equal to (1-α),

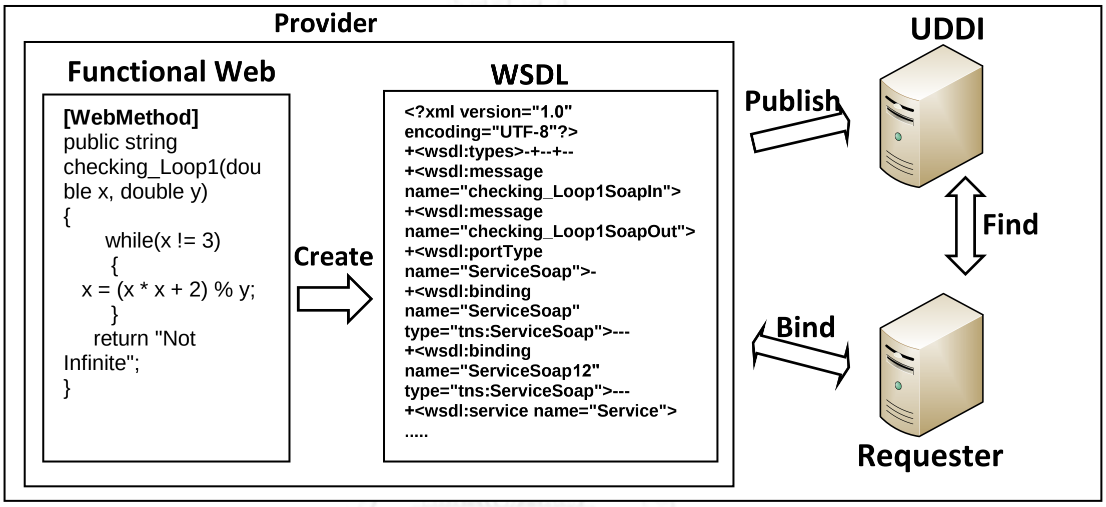
s is the standard variation of sample execution time values during time *t*, and

n is the total number of the executions during time *t*.

2.4 Web Services

A web service is software that grants services over the web technology. A web service is an improvement on Remote Procedure Calls (RPC) or methods for exchanging the information using the SOAP protocol among the business processes. One important characteristic of web service algorithms is the same as any other software, which is the reliability of the web service process. Thus, each web-service module must be tested for reliable services [26-29] before being used in the real world.

Functional web services are the functional programs which are implemented at the web-service providers. They will be executed after receiving the SOAP request message. Figure 2.2 shows the basic structure of web services and functional web services.



```
[WebMethod]
public string
checking_Loop1(dou
ble x, double y)
{
      while(x != 3)
       {
   x = (x * x + 2) % y;
       }
     return "Not
Infinite";
}
```

```
<?xml version="1.0"
encoding="UTF-8"?>
+<wsdl:types>-+--+--
+<wsdl:message
name="checking_Loop1SoapIn">
+<wsdl:message
name="checking_Loop1SoapOut">
+<wsdl:portType
name="ServiceSoap">-
+<wsdl:binding
name="ServiceSoap"
type="tns:ServiceSoap">---
+<wsdl:binding
name="ServiceSoap12"
type="tns:ServiceSoap">---
+<wsdl:service name="Service">
.....
```

Figure 2.2 The basic structure of web services and functional web services

2.4.1 The Run-time Error Detection of Web Services

Normally, web services must pass the verification and validation (V&V) processes as same as any programs. V&V are the main techniques for confirming correctness on the objective of general software services. So, there are many V&V models designed for proving the completeness of the functions [4-5]. Error detection was covered on the environment at run-time. Then, the system should be protected from these errors by creating a framework, a model or architecture for monitoring and detecting run-time faults. Run-time errors may occur on many factors at run-time, such as incorrect inputs, file not found, or the fault from the system's files.

Generally, based on the run-time error detection, there are three main solving engines: the fault detector, replication, and notification, shown as Figure 2.3.

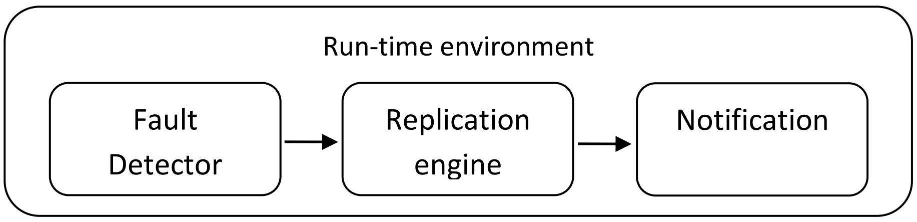


Figure 2.3 The three main engines for solving errors on the run-time error detecting

Referring to Figure 2.3, each module can be described as follows:

Fault detector: this engine is responsible for checking run-time errors. Typically, the run-time errors are captured by the exception classes. Thus, the monitoring technique based on detecting faults are classified by tools or solving techniques [9-30].

Replication engine: this engine is responsible for system recovery when a run-time error occurs. The replication technique may use many services in many servers in the distributed system. Nevertheless, the replication technique may cause high system overhead depending on the number of replication. Some errors cannot use replication technique in the same engine such as the server cannot respond, and file-not-found error etc.

The errors on the recovery engine are separated into two types: errors that cannot use the replication technique, and errors that can use the replication technique. Based on the errors that cannot use the replication technique, the developers must be sure that the input data in the testing process cannot lead to this type of error again, which is different from the second type of the error, in that the same input data can be reused. In the distributed system such as web services system, the replication technique is used in fault-tolerant systems [31-34].Moreover, the replication technique was proposed with separation into active replication and passive replication [35-36].

Notification engine: this engine is responsible for informing the results to the requester after recovery.

In the next chapter, Chapter 3, the infinite loop situation will be considered and described more in detail. In addition, the proposed mechanism to detect and solve the infinite loop situation under run time with the implementation of the EMTI value will be elaborated.

CHAPTER 3
DESIGN OF THE MECHANISM

This chapter describes the design of the detection and control mechanism for infinite-loop situations. To detect the infinite-loop situation, the loop condition and iteration variables are considered. The proposed solution for the design can be separated into three sections. In the first section, loop characteristics are analyzed to identify the loop conditions and iteration variables. The technique for defining the execution time boundary is proposed in the second section. In the third section, the analysis of the non-deterministic loop structure is studied to detect the infinite-loop situation. In addition, the following section provides examples of different loop types. Furthermore, the designed mechanism of the proposed solution is described in the last section.

3.1 Loop Characteristics

According to the definitions of loops mentioned in Chapter 1, this section provides more details of two important loop types: deterministic loops and non-deterministic loops. However, there are three significant factors that will be focused on in this section:

Iteration variable (iv)[3] is a variable relating to the loop condition. If the loop condition is TRUE, the next loop cycle will be executed. The value of *iv* is set in two phases: the initial phase and the loop cycle phase.

- Initial phase: the value of *iv* is set before entering the loop instruction.
- Loop cycle phase: the value of *iv* is modified for the next loop cycle. Thus, there are some statements that change the *iv* value, which means adjusting values of the iteration variable each loop cycle, called *incr_expr*. In each loop cycle, the *iv* value will be compared with the loop condition expression. The loop will be terminated whenever the output of the loop condition is FALSE.

Exit loop condition (exit_cond) [3] is a set of statements that terminate the loop. The *exit_cond* relates to two variables: *iv* and *x*, where *x* is a value that is used to control the execution cycles of a loop. The types of statement for the exit loop condition are as follows:

1. "*iv*<= *x*" : the loop condition runs when *iv* is less than or equal to *x.* On the other hand, the loop condition will be terminated when *iv* is greater than *x*.
2. "*iv*<*x*" : the loop condition runs when *iv* is less than *x.* On the other hand, the loop condition will be terminated when *iv* is greater than or equal to *x*.
3. "*iv*>= *x*" : the loop condition runs when *iv* is greater than or equal to *x.* On the other hand, the loop condition will be terminated when *iv* is less than *x*.
4. "*iv*>*x*" : the loop condition runs when *iv* is greater than *x.* On the other hand, the loop condition will be terminated when *iv* is less than or equal to *x*.

Adjusting values of iv (incr_expr)[3] is a set of statements for changing the *iv* value in each loop cycle. The *incr_expr* relates to two variables: *iv* and *x,* where *x* is a value that is used to control the execution of a loop.

The *incr_expr* of a loop consists of four statements as follows.

1. "++*iv*' and '*iv*++" : *iv* value will be incremented by one before and after executing the statement, respectively.
2. "--*iv*' and '*iv*--" : *iv* value will be decremented by one before and after executing the statement, respectively.
3. "*iv* += *x*", "*iv* = *iv* + *x*" and "*iv* = *x* + *iv*" : *iv* value will be incremented by *x* value.
4. "*iv* -= *x*" and "*iv* = *iv* - *x*" : *iv* value will be decremented by *x* value.

3.1.1 Deterministic loop

Referring to the loop characteristics in Chapter 1, both the static loop and the countable loop are deterministic loops. One characteristic of the countable loop is that a number of loop cycles can be guaranteed and the infinite-loop situation can be predicted. Table 3.1 defines the components of all countable loops.

Table 3.1 Defining components of countable loop [3]

Examples of loops	Definitions apply	Restriction
for (initial [*iv*]; *exit_cond*;	***exit_cond*:**	1. the *exit_cond* is not
incr_expr)		affected from statements
statement	*iv <= ub*	inside or outside of the loop.
....	*iv <ub*	2. the *incr_expr* expression is
for (initial [*iv*]; *exit_cond*;	*iv >= ub*	not within a critical section.
incr_expr) {	*iv >ub*	3.***iv*** : Iteration variable. The
[*declaration_list*]		iteration variable is a signed
[*statement_list*]	***incr_expr*:**	integer that has either
}		automatic or register storage
....	*++iv*	class, does not have its
Initial [*iv*];	*iv++*	address taken, and is not
while (*exit_cond*) {	*--iv*	modified anywhere in the
[*declaration_list*]	*iv--*	loop except in incr_expr.
[*statement_list*]	*iv += incr*	
incr_expr;	*iv -= incr*	4.***incr*** : Loop invariant signed
[*statement_list*]	*iv = iv + incr*	integer expression. The value
}	*iv = incr + iv*	of the expression is known at
....	*iv = iv - incr*	compile-time and is not 0.
Initial [*iv*];		incr cannot reference extern
do {		or static variables, pointers or
[*declaration_list*]		pointer expressions, function
[*statement_list*]		calls, or variables that have
incr_expr;		their address taken.
[*statement_list*]		5.***ub*** : Loop invariant signed
} while (*exit_cond*)		integer expression. ub cannot
		reference extern or static
		variables, pointers or pointer
		expressions, function calls, or
		variables that have their
		address taken.

From the structure of loop examples in Table 3.1, a number of loop cycles can be predicted. In the deterministic loop, the loop execution time can be predicted with certainty. An example of a deterministic loop is presented in Figure 3.1.

```
main()
{         int x = 0;
          while (x <= 99)
          {   printf("x = %d\n", x);
          x++;     }
}
```

Figure 3.1 An example of a deterministic loop algorithm

3.1.2 Non-Deterministic loop

As mentioned in Chapter 1, in the non-deterministic loop the number of loop cycles is unknown, and the loop execution time cannot be predicted with certainty.

Referring to *iv* mentioned in Section 3.1, non-deterministic loops with the *exit_cond* and *incr_expr* are divided into two sections: a critical section and a non-critical section.

In the non-deterministic loop with a non-critical section, the variables and set of statements in *exit_cond* and *incr_expr* are the same as the countable loop which is different on restrictions.

In the non-deterministic loop with a critical section, *exit_cond* mostly only has only one expression: “iv != constant value”. This means loop termination has only one condition. Moreover, *incr_expr* cannot be clearly defined in calculation statement for changing *iv* value in each loop cycle.

Therefore, a non-deterministic loop with non-critical sections has lesser chance of being the infinite loop than with critical sections, because the non-critical section easily performs in *incr_expr* calculation. Therefore, for the infinite-loop situation, only the non-deterministic loop with the critical section should be considered. If both *exit_cond* and *incr_expr* fall into the critical section, the opportunity of an infinite-loop situation occurring is high.

```
main()
{          int x = 0, n;
           scanf("%d", &n);
           while (x <= n)
           {   printf("x = %d\n", x);
           x++; }
}
```

Figure 3.2 Example of a non-deterministic loop algorithm

3.2 Characteristic Analysis of Critical Section

Consider the following example, Figure 3.3, in which the values of *x* and *y* are dependent on other functions, *foo()* and *bar()*. Thus, there is a high possibility that the *exit_cond* cannot be satisfied when running.

```
main()
{          x = foo();
           y = bar();
           while (x != 3)
           {   x = (x * x + 2) % y; }
}
```

Figure 3.3 Example of a non-deterministic loop algorithm with critical section with repetition of *iv* values [17]

Based on the previously defined definition, the *iv* is *x*. The 'exit_cond' is "*x* != 3" and *ub* is a constant value, 3. Therefore, the loop will be terminated whenever *x* is equal to 3. While the value of *y* will never be changed in the loop, the value of *x* is changed according to the statement "*x* = (*x* * *x* + 2) % *y*" inside the loop.

Assuming that *foo()* returns 1 and *bar()* returns 2, the initial *x* value is set to 1 and the initial *y* value was it to 2. After passing the first cycle, *x* has the same value because the calculation of statement "*x* = (*x* * *x* + 2) % *y*" cannot change the value of x. Therefore, if *x* is equal to 1 and *y* is equal to 2, the loop cannot be terminated because *x* value cannot be equal to 3 and the infinite loop will occur.

In another case, if *foo()* returns 7 and *bar()* returns 5,the initial *x* value is set to7 and the initial *y* value is set to 5. After passing the first cycle, the value of *x* is 1 and the next cycle, the value of *x* is 3. Thus, the loop can terminate after passing the second cycle of the loop.

From preliminary data analysis of this loop, it can be concluded that the loop cannot be terminated whenever value of *x* and *y* consist of:

- An initial *y* value between 1 and 3, because *x* value cannot be 3 with the fraction, which is divided by *y* value.
- An even initial *y* value and an even initial *x* value.

On the other hand, the loop can be terminated when the *x* and *y* data consist of;

- An even initial *y* value and an odd initial *x* value.
- An odd initial *y* value and almost all initial *x* values both even and odd. This case has some *x* values that can cause non-termination of the loop.

As mentioned above, *y* is not related to the loop instruction, but *x* is *iv*. Therefore, *x* values should be considered when the loop cannot be terminated.

In the execution phase, when *foo()* returns 6 and *bar()* returns 10, in every loop cycle, the value of *x* is changed only to 6 and 8, a set of repeating values, as shown in Table 3.2. Thus, the loop condition cannot be FALSE and the infinite loop will occur.

Table 3.2 An example of repetition of *iv* values

Loop Cycle	1	2	3	4	5	6	7	8	9	10	…	n-1	n
iv values (x)	6	8	6	8	6	8	6	8	6	8	…	6	8

Another example of a non-deterministic loop algorithm is shown in Figure 3.4.

```
main()
{
floatx = 0.1;
while (x != 1.1) {
printf("x = %f\n", x);
        x = x + 0.3;      }
}
```

Figure 3.4 An example of non-deterministic loop algorithm in critical section with the growth of *iv* values

From the algorithms of Figure 3.4, the algorithm has one loop instruction, that is, “while (*x* != 1.1)”. *iv* is x. The initial *iv* value is set to a constant value, 0.1. The *x* value is changed on a calculating statement in the loop: “*x* = *x* + 0.3”. Therefore, the loop condition cannot be true because the *x* value cannot be equal to value 1.1. In each loop cycle, the value of *x* will be added continuously and cannot exit the loop. The infinite loop always occurs whenever this algorithm is running. Thus, the fault occurs from the structure of algorithm.

On the other hand, if the initial *iv* value is not a constant value, the loop can be terminated in some cases of the initial *iv* values, such as 0.8, 0.5, etc. So, when an infinite loop occurs and *iv’s* value is continuously increased until it is higher than the *ub* value, as shown in Table 3.3.

Table 3.3 An example of the growth of *iv* values

Loop cycle	1	2	3	4	5	6	7	8	9	10	...	N
iv values (x)	0.1	0.4	0.7	1	**1.3**	1.6	1.9	2.2	2.5	2.8	...	...

higher than *ub* value

As mentioned above, the pattern of *iv* values under the infinite loop situation can be defined in Definition 4 to Definition 8 as follows:

Definition 4:

Let $P_{i/n}$ be a series of numbers with length n starting at position i in *iv* values and can be defined as $P_{i/n} = P_{i1}P_{i2} \dots P_{in}$ where P_{ik} is the number of $P_{i/n}$ at digit k.

Definition 5:

Let $P_{i/n}$ and $P_{j/n}$ be a series of numbers with length n starting at position i and j in *iv* values. The *iv* values repeat if and only if $P_{i/n}$ and $P_{j/n}$ in every digit of $P_{i/n}$ and $P_{j/n}$, that is, $P_{i1} = P_{j1}, P_{i2} = P_{j2}, P_{i3} = P_{j3}, \dots, P_{in} = P_{jn}$

Definition 6:

Let $P_{i/n}$ be a series of numbers with length n started at position i in *iv* values. The *iv* values monotonically increase if and only if, $\forall\, i < j, P_{i/n} < P_{j/n}, \exists\, P_{i/n}$.

Definition 7:

Let $P_{i/n}$ be a streaming number with length *n* started at position *i* in *iv* values. The *iv* values monotonically decrease if and only if $\forall\, i < j, P_{i/n} > P_{j/n}, \exists\, P_{i/n}$.

Definition 8:

The *iv* values has a *pattern* if and only if $P_{i/n}$in *iv* values is satisfied either a condition stated in Definition 4, 5, 6, or 7.

By analyzing data defined in Definitions 4 – 8, the pattern would occur in two cases: 1) the *iv* values are repeated and 2) the *iv* values increase or decrease continuously, which will be mentioned in Chapter 4.

Next, Classification of Problem Domain will be described by designing algorithms with non-deterministic loops with a critical section.

3.3 Classification of Problem Domain

Since the consideration of loops in this research is based on three types of C language command and there are three infinite loop patterns as mentioned above, nine cases of web services are deployed in this experiment as listed in Table 3.4.

Table 3.4 Examples of web service modules with non-deterministic loops

Loop types	NO.	Web service modules

while	1	[WebMethod]
		public string Calculation1 (double x, double y)
		{
		        while ((x != 3) && (y > 0))
		        { x = (x * x + 2) % y;}
		    return "No Infinite Loop";
		}

	2	[WebMethod]
		public string Calculation2 (double x)
		{
		        while (x != 1.1)
		        {x = x + 0.3;}
		    return "No Infinite Loop";
		}

	3	[WebMethod]
		public string Calculation3(double x)
		{
		        while (x != -2.1)
		        { x = x - 0.2; }
		    return "No Infinite Loop";
		}

for	4	[WebMethod]
		public string Calculation4 (double n, double y)
		{
		   for (double x = n; (x!=3) && (y>0); x = (x * x + 2) % y)
		    { }
		   return "No Infinite Loop";
		}

	5	[WebMethod]
		public string Calculation5 (double n)
		{
		  for (double x = n; x!=1.1; x=x+0.3)
		  { }
		  return "No Infinite Loop";
		}

Table 3.4 Examples of web service modules with non-deterministic loops (con.)

Loop types	NO.	Web service modules
for	6	[WebMethod]
		public string Calculation6 (double n)
		{
		for (double x = n; x!=-1.1; x=x-0.3)
		{ }
		return "No Infinite Loop";
		}
do…while	7	[WebMethod]
		public string Calculation7 (double x, double y)
		{
		do
		{ x = (x * x + 2) % y;
		} while ((x != 3) && (y > 0));
		return "Not Infinite";
		}
	8	[WebMethod]
		public string Calculation8 (double x)
		{
		do
		{ x = x + 0.3;
		} while (x != 1.1);
		return "Not Infinite";
		}
	9	[WebMethod]
		public string Calculation9(double x)
		{
		do
		{ x = x - 0.2;
		} while(x != -2.1);
		return "No Infinite Loop";
		}

In the next section, the mechanism for detecting the infinite-loop situation is proposed from previous investigation.

3.4 The proposed Mechanism

A mechanism is proposed for detecting the infinite-loop situation from the boundary of execution time and the *iv* values. The mechanism is performed using the activity diagram. The activity diagram of the mechanism is separated into four diagrams: activity diagram for monitoring web service module, activity diagram for training the IWSM, activity diagram for verifying the IWSM, and activity diagram for online-verifying the IWSM. Each diagram will be described as follows:

3.4.1. Activity Diagram for Monitoring Web Service Module

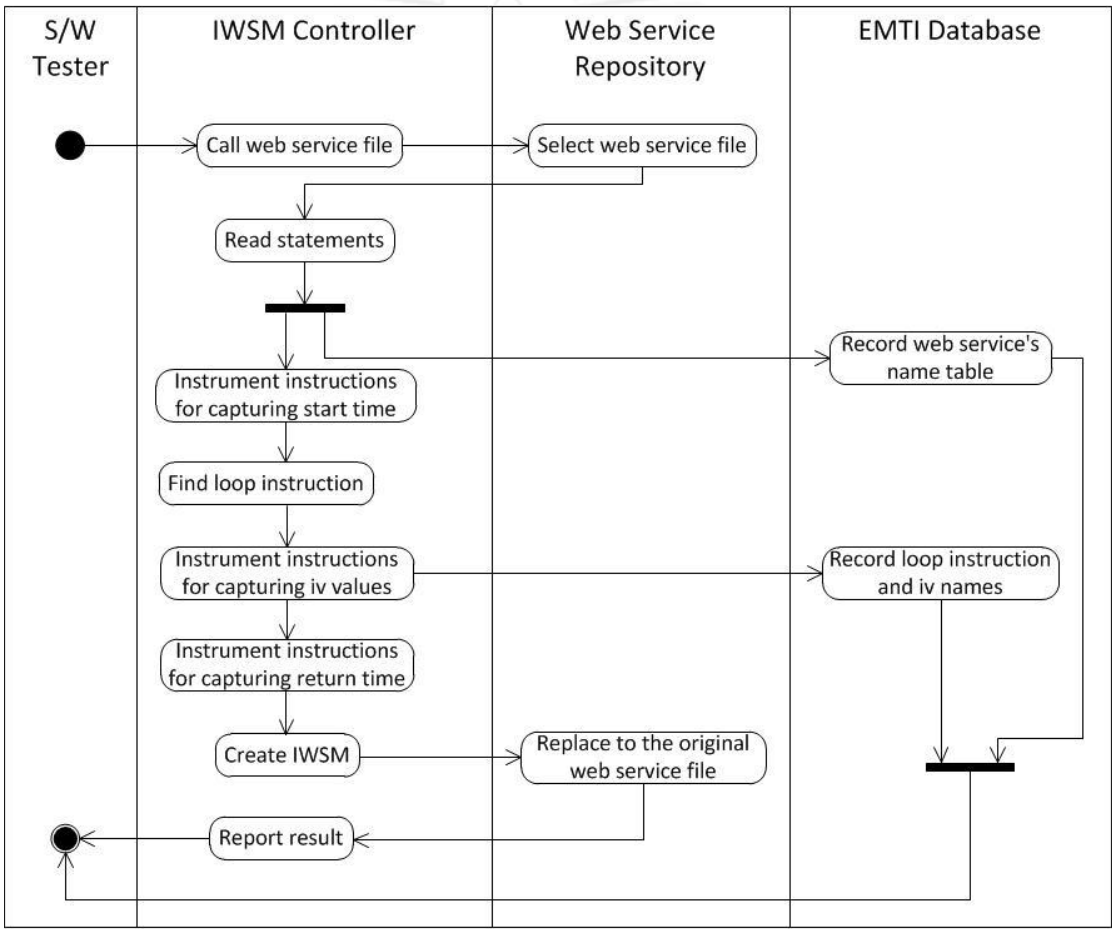

S/W Tester
IWSM Controller
Web Service Repository
EMTI Database
Call web service file
Select web service file
Read statements
Record web service's name table
Instrument instructions for capturing start time
Find loop instruction
Instrument instructions for capturing iv values
Record loop instruction and iv names
Instrument instructions for capturing return time
Create IWSM
Replace to the original web service file
Report result

Figure 3.5 Activity diagram for monitoring web service module

Figure 3.5 shows the activity diagram of the instrumented instructions. There are four related classes: S/W tester, IWSM controller, Web service repository, and EMTI database. This diagram starts as the S/W tester chooses the web service file for the instrumented instructions at the IWSM controller. The IWSM controller calls the web service file from the web service repository. The web service repository sends the web service file to the IWSM controller. The IWSM controller reads each statement in the web service file. First, in the header of web service statements, the instructions will be instrumented for capturing the start time, and the web service name will be used for creating the web service name table at the EMTI database. Secondly, when a loop instruction is found, the loop body will be instrumented with instructions for capturing *iv* values, and the loop instruction and the *iv* names will be recorded in the EMTI database. Lastly, instruction instrumentation for capturing the return time will be done before returning the statement. After passing the instrumented instruction in each step, the instrumented web service module (IWSM) is created in the IWSM controller and it will overwrite the original web service file in the web service repository.

After that, the IWSM is ready to be used for detecting the infinite-loop situation. In the first step, the IWSM will be trained with an input data training set, which is the numbers of input parameters that will not cause an infinite-loop situation. The activity diagram for training the IWSM is shown in Figure 3.6.

3.4.2. Activity Diagram for Training the IWSM

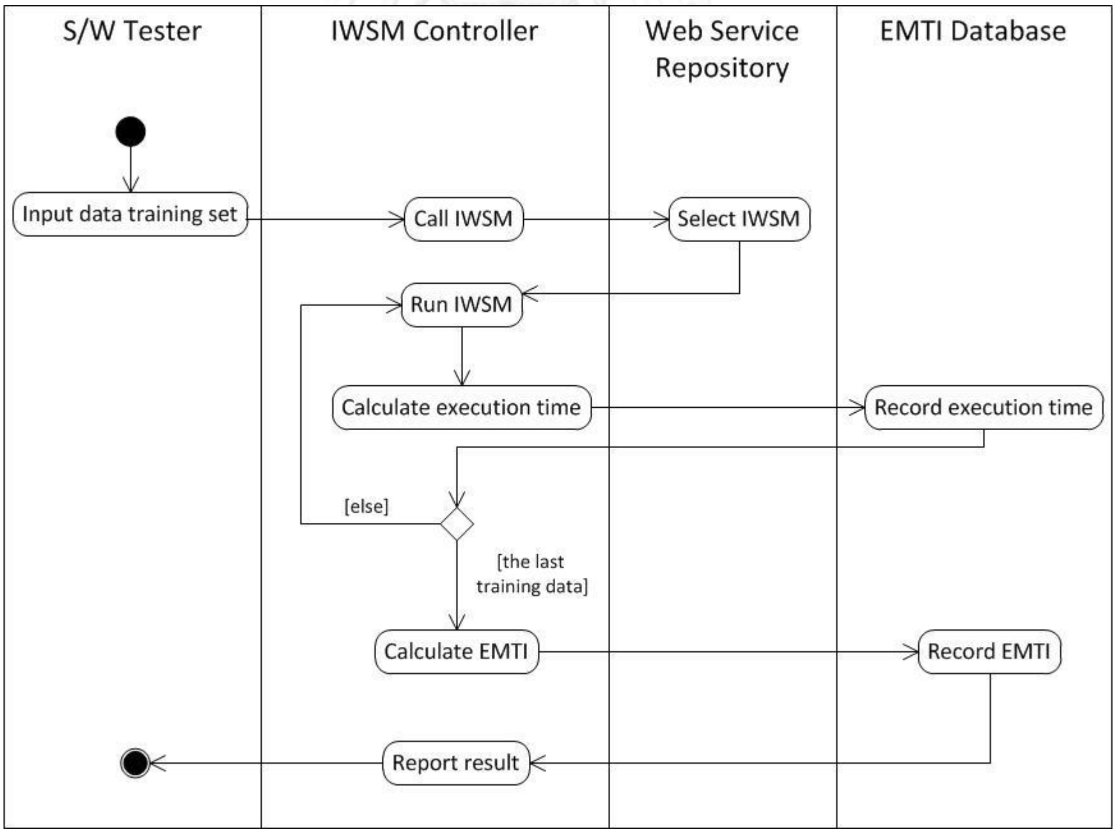


Figure 3.6 Activity diagram for training the IWSM

Referring to Figure 3.6, the activity diagram for training the IWSM is separated into four classes: S/W tester, IWSM controller, Web service repository, and EMTI database. This diagram starts as the S/W tester sends the input data training set. The IWSM controller calls an IWSM, and the IWSM will be selected by the web service repository. Whenever the IWSM is run with each input data training value, the execution time will be calculated and recorded in the EMTI database. If the input data training value is not the last value, the IWSM will be called continuously. On the other hand, if the input data training value is the last value, the EMTI value will be calculated from the execution time values and recorded at the EMTI database. The initial EMTI value will be calculated and recorded in the EMTI database. After the EMTI value is recorded, the result report will be created for informing the S/W tester.

After passing this step, the initial EMTI value will be calculated from a number of execution times from the input data training set. In the next step, the IWSM will be tested with the input data test set. The data test set cannot guarantee that the infinite-loop situation will not occur. Thus, for some data elements in the data test set, the infinite-loop situation may arise. The activity diagram for verifying the IWSM is shown in Figure 3.7.

3.4.3. Activity Diagram for Verifying the IWSM

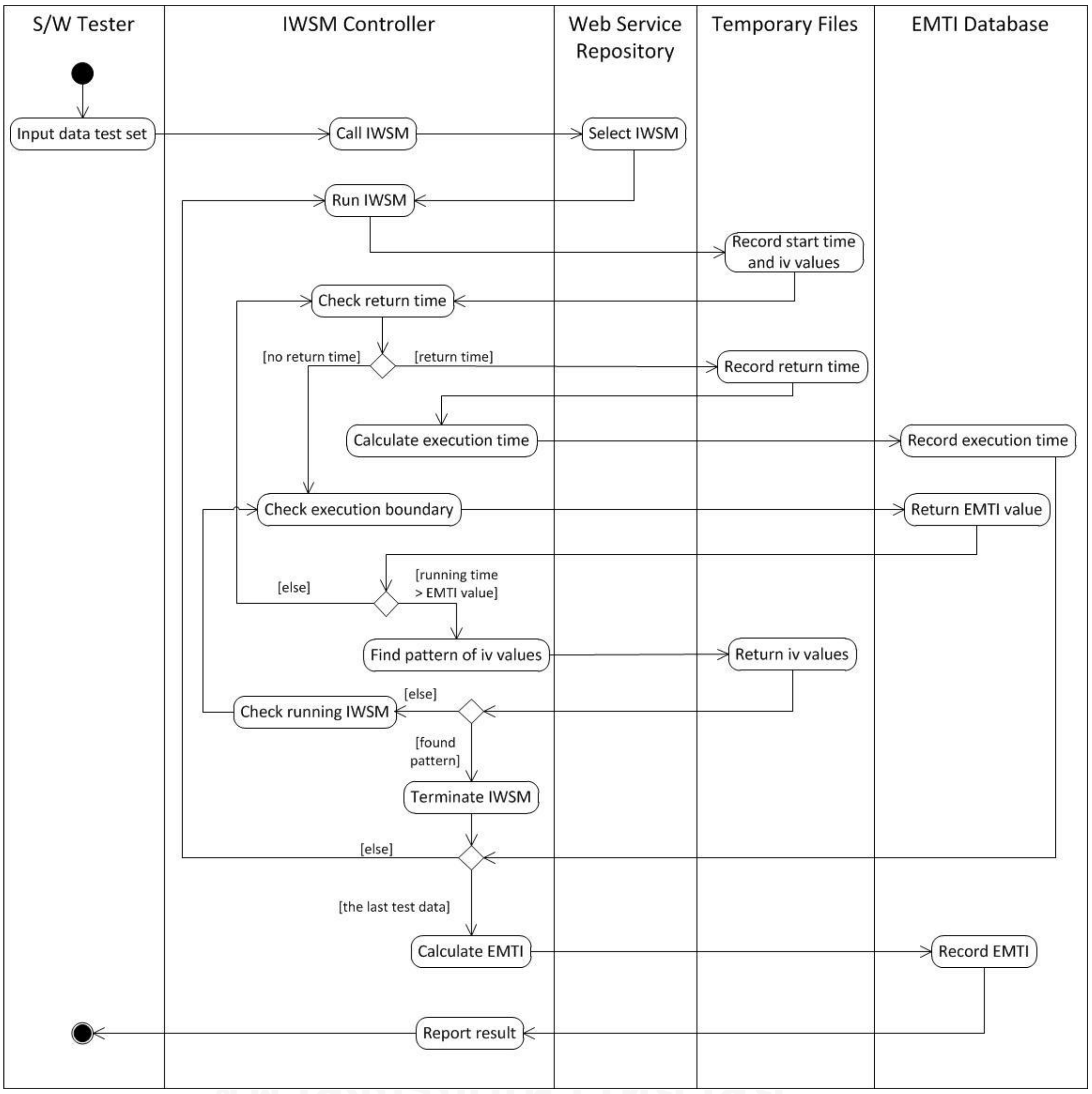


Figure 3.7 Activity diagram for verifying the IWSM

Figure 3.7 shows the activity diagram for verifying the IWSM, which is separated into five classes: S/W tester, IWSM controller, Web service repository, temporary files and EMTI database. This diagram starts as the S/W tester sends the input data test set. The IWSM controller calls the IWSM, which will be selected from the web service repository. When the IWSM is run, the start time and the *iv* values will be recorded in the temporary files. The IWSM controller checks the return time value. If the return time has a value, the IWSM controller calculates the execution time and records the execution time in the EMTI database. On the other hand, if the return times do not have a value, the IWSM controller will check the execution boundary of the IWSM by using the EMTI value from the EMTI database. If the current running time value of the IWSM is greater than the EMTI value, the IWSM controller will search for a pattern of *iv* values from the temporary file. If a pattern is found, the infinite-loop situation is also found, and the IWSM will be terminated by the IWSM controller. In contrast, if the pattern is not found, the IWSM controller will check the execution of the IWSM and the execution boundary time again.

If the input data test value is not the last value, the IWSM will be called continuously. On the other hand, if the input data test value is the last value, the new EMTI value will be calculated from the execution time values and recorded in the EMTI database. After the new EMTI value is recorded, the result report will be created for informing the S/W tester.

3.4.4. Activity Diagram for Online-Verifying the IWSM

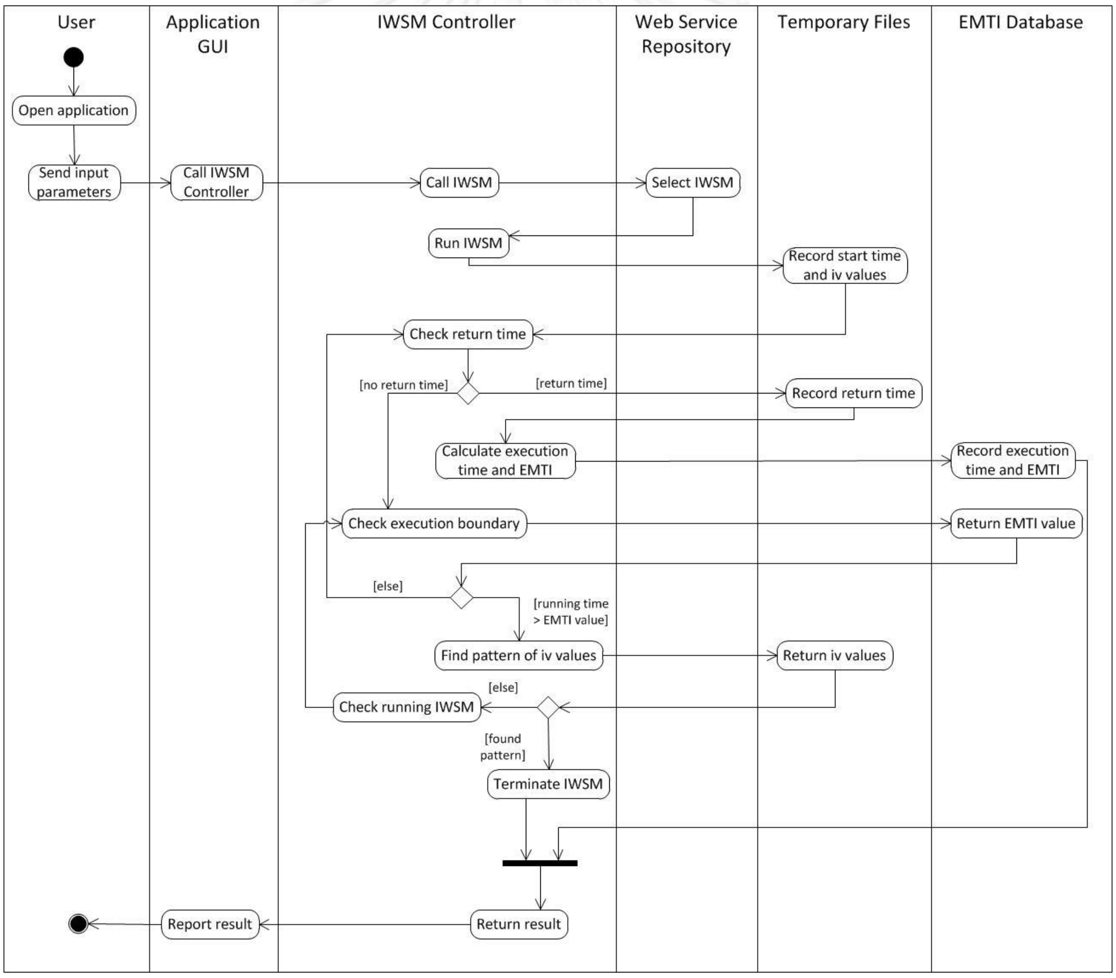


Figure 3.8 Activity diagram for Online-Verifying the IWSM

The activity diagram for online-verifying the IWSM is similar to the activity diagram for verifying the IWSM, but there are two additional classes: User and Application GUI. This diagram starts as the user opens the application and sends the input parameters to the application GUI. Thus, the application GUI calls the IWSM controller. Again, the process is similar to the previous one, but a new EMTI value will be calculated every time. The IWSM controller receives the execution time value instead of being calculated only after the IWSM controller receives the set of execution time values from the EMTI database.

3.5 The Algorithm for Creating the IWSM

Referring to Figure 3.5, the activity diagram for monitoring web service modules can be presented as pseudo code in Figure 3.9. This pseudo code shows the concepts that are applied to instrumenting detection code into a web service.

```
Read file stream of web service module
DOWHILE(NOT End Of File)
  read line
  IF (line = web service header instruction) THEN
   find web service name
   create web service name table to EMTI database
   instrument code for capturing start time
  ENDIF
  IF (line = loop instruction) THEN
   find iteration variables
   instrument code for capturing iteration variable values
into loop body
  ENDIF
  IF (line = return instruction) THEN
   instrument code for capturing return time
  ENDIF
ENDDO
```

Figure 3.9 Algorithm for creating the instrumented web service module

3.6 The Algorithm for Verifying the IWSM

Since the boundary time and pattern finding are the very significant solution of this research, the algorithms of both processes are presented in Figure 3.10. In addition, these algorithms are implemented in both the verification and the online-verification of IWSM.

```
set s1 to null
set t1 to 0
set t2 to 0
set b1 to false

While (time for checking = true)
For each IWSM's name
  s1 = IWSM's name
  get EMTI values from EMTI_DB database from s1
  t2 = upper boundary of EMTI values
  get start and return time values of s1 from file storage
  t1 = start time value + t2

  IF (return time value = empty)
    IF (current time of the system > t1)
     For each iv values files of s1
      get iv values from iv values files
      b1 = Find pattern (iv values)
       IF (b1 = true)
         call Terminate IWSM(s1)
       ENDIF
     ENDFOR
   ENDIF
 ENDIF
ENDFOR
ENDWHILE

Find pattern(iv values)
 IF (repetition of iv values = true)
     return TRUE
 ELSE IF (increasing of iv values = true)
     return TRUE
 ELSE IF (decreasing of iv values = true)
     return TRUE
 ELSE  return FALSE
 ENDIF
END FUNCTION

Terminate IWSM(s1)
     destroy process name s1
     create report
END FUNCTION
```

Figure 3.10 The algorithm for checking boundary times and finding patterns of *iv* values

In the next chapter, Chapter 4, all implementation details will be shown as a deployment diagram. Moreover, the information on testing the proposed method and architecture is presented to ensure that the infinite-loop situation can be detected.

CHAPTER 4
IMPLEMENTATION AND EXPERIMENTAL RESULTS

This chapter discusses details of the implementation and experiments for evaluating the proposed mechanism.

4.1 The Proposed Architecture

The proposed architecture is shown in the deployment diagram in Figure 4.1 where components of the architecture are defined.

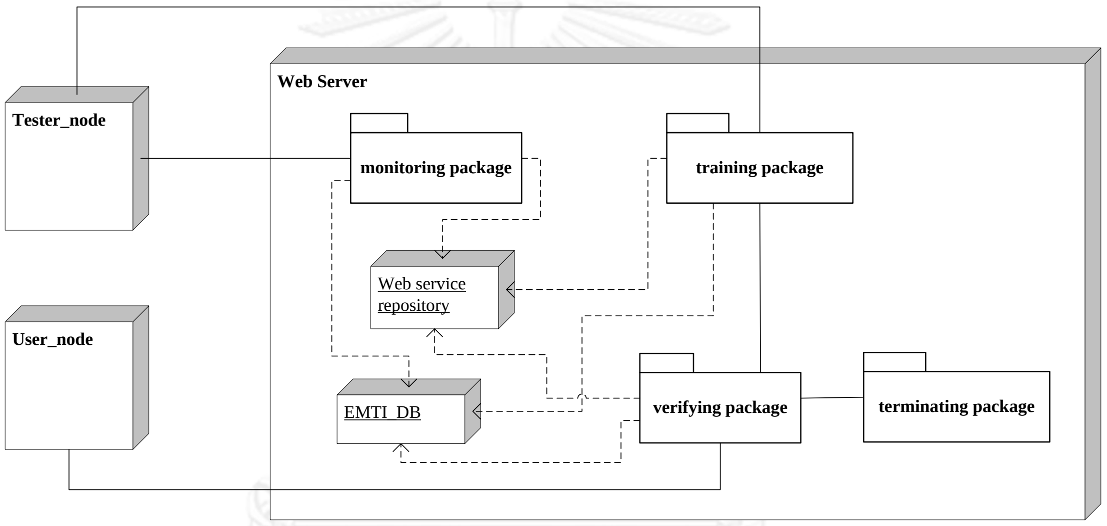


Figure 4.1 Deployment diagram of proposed architecture

According to the designed mechanism in Chapter 3, Figure 4.1 shows the deployment diagram of the derived system that is installed on a web server. Based on the deployment diagram, there are four packages that must be processed in this approach: the monitoring package, training package, verification package, and termination package. In addition, two databases are involved: the web service repository and EMTI_DB. Details of each package and database will be described in the following sections.

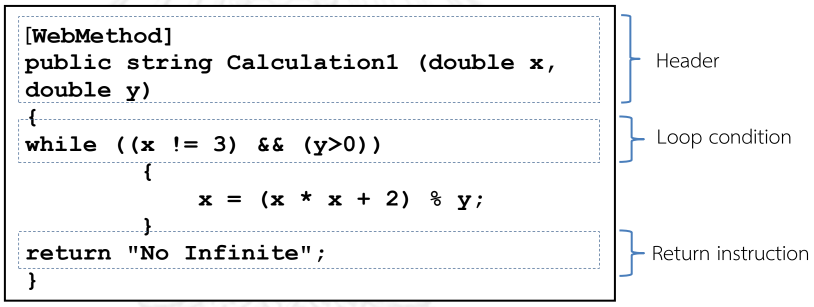
4.1.1 Monitoring Package

This package is used by S/W testers who have the responsibility to verify the web services in the server. One important module is the Instruction_Instrument module since it is responsible for instrumenting the infinite loop monitoring code.

4.1.1.1 Instruction_Instrument Module

Whenever a web service is implemented, the S/W tester will read this web service agent as an input to this module. The output of this step is the IWSM.

Referring to Definitions 4 - 8 in Chapter 3, this research analyzes the changing of *iv* values in the loop implementation. Therefore, this module is responsible for instrumenting instructions for recording *iv* values and capturing the execution time, which refers to the algorithms for creating the IWSM in Chapter 3.



```
[WebMethod]
public string Calculation1 (double x,
double y)
{
while ((x != 3) && (y>0))
          {
               x = (x * x + 2) % y;
          }
return "No Infinite";
}
```

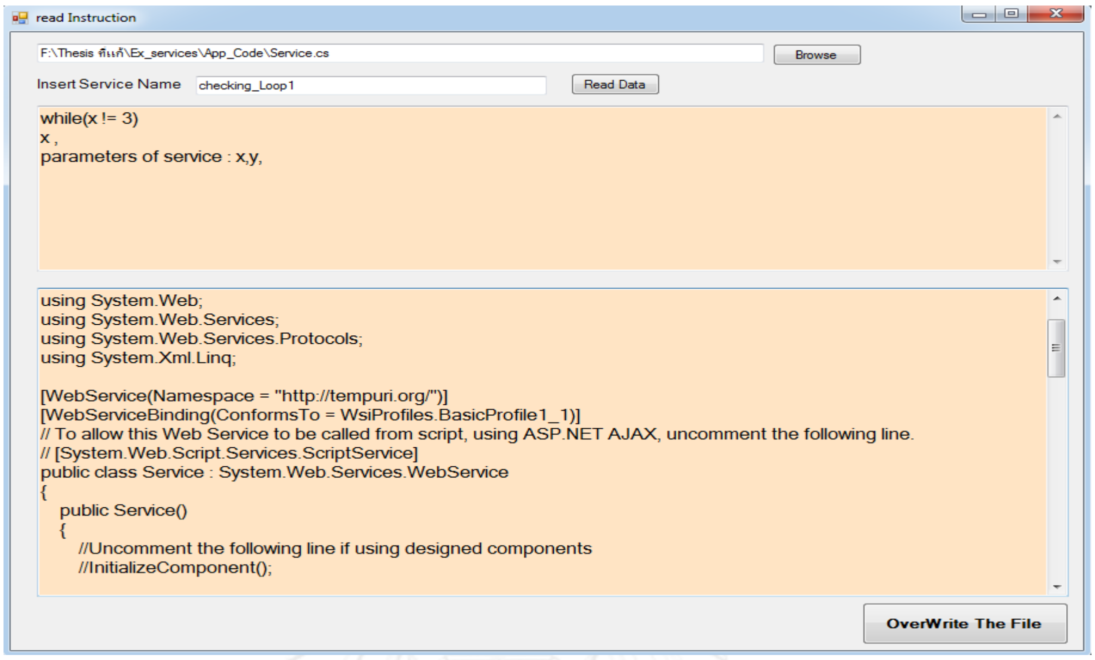
Figure 4.2 Three parts of finding instructions

When considering an implemented web service module, there are three parts to be considered: header, loop condition, and return instructions, as shown in Figure 4.2.

- Header : To find the web service module name and input parameters of the service.
- Loop condition : To find the *iv* of the loop.
- Return instructions : To find the return instruction.

After instrumenting instructions, the structure will be recorded in the EMTI_DB and the IWSM will be overwritten in the same file. The application for the Instruction_Instrument module is shown in Figure 4.3. Moreover, Figure 4.4 shows the original web service before the instructions were instrumented by the

Instruction_Instrument module and Figure 4.5 shows the web service algorithm after the instructions have been instrumented.



F:\Thesis ที่แก้\Ex_services\App_Code\Service.cs

Insert Service Name: checking_Loop1

```
while(x != 3)
x ,
parameters of service : x,y,
```

```
using System.Web;
using System.Web.Services;
using System.Web.Services.Protocols;
using System.Xml.Linq;

[WebService(Namespace = "http://tempuri.org/")]
[WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]
// To allow this Web Service to be called from script, using ASP.NET AJAX, uncomment the following line.
// [System.Web.Script.Services.ScriptService]
public class Service : System.Web.Services.WebService
{
    public Service()
    {
        //Uncomment the following line if using designed components
        //InitializeComponent();
```

Figure 4.3 An example of the Instruction_Instrument module

```
public class Service : System.Web.Services.WebService
{
    public Service()
    {     }

    [WebMethod]
    public string Calculation1(double x, double y)
    {
        while ((x != 3) && (y > 0))
        {
            x = (x * x + 2) % y;
        }
        return "No Infinite";
    }
}
```

Figure 4.4 The original web service before instrumenting instructions

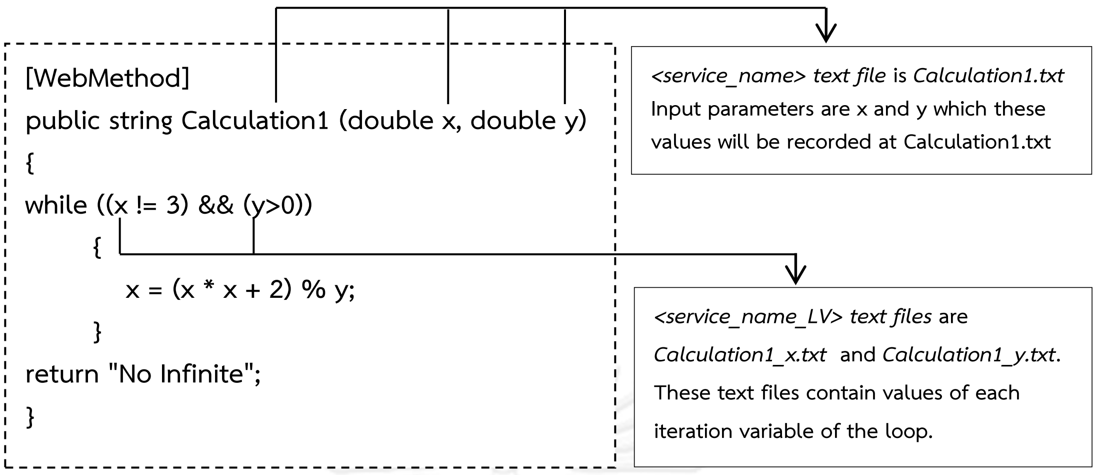
```
public class Service : System.Web.Services.WebService
{
    public Service()
    {    }

    DateTime t1, t2;
    Boolean bool1 = true;
    Boolean bool2 = true;
    string x0;
    string y1;
    [WebMethod]
    public string Calculation1(double x, double y)
    {
        GC.Collect();
        TimeSpan d1 = new TimeSpan();
        d1 = DateTime.Now.TimeOfDay;
        StreamWriter T_d1;
        T_d1 = File.AppendText(@"D:\EMTI_ARCH_results\Calculation1.txt");
        T_d1.WriteLine("");
        T_d1.Write(" x " + x + " y " + y + " " + d1 + " ");
        T_d1.Close();
        OleDbConnection TimeCon1 = new
OleDbConnection(@"Provider=Microsoft.ACE.OLEDB.12.0; Data
Source=D:\EMTI_DB.accdb");
        TimeCon1.Open();
        OleDbCommand com = new OleDbCommand("select max_exeTime from EMTI
where Service_name='Calculation1'", TimeCon1);
        OleDbDataReader myReader = com.ExecuteReader();
        while (myReader.Read())
        {
            t1 = DateTime.Now;
            t2 =
t1.AddMilliseconds(Convert.ToDouble(myReader.GetValue(0).ToString()));
        }
        myReader.Close();

        while ((x != 3) && (y > 0))
        {
            if ((DateTime.Now >= t2) && (bool1 == true))
            {
                get_data0(x0);
                x0 = null;
                bool1 = false;
            }
            x0 += x + "\\\\";
```

Figure 4.5 The output from the Instruction_Instrument module, the IWSM

```
            if ((DateTime.Now >= t2) && (bool2 == true))
            {
                get_data1(y1);
                y1 = null;
                bool2 = false;
            }
            y1 += y + "\\\\";
            x = (x * x + 2) % y;
        }
        TimeSpan d2 = new TimeSpan();
        d2 = DateTime.Now.TimeOfDay;
        StreamWriter T_d2;
        T_d2 = File.AppendText(@"D:\EMTI_ARCH_results\Calculation1.txt");
        T_d2.Write(d2 + "*");
        T_d2.Close();
        File.Delete(@"D:\EMTI_ARCH_results\Calculation1.x.txt");
        File.Delete(@"D:\EMTI_ARCH_results\Calculation1.y.txt");
        return "No Infinite";
    }
    void get_data0(string str)
    {
        StreamWriter Tx;
        Tx =
File.CreateText(@"D:\EMTI_ARCH_results\Loop_Variable_Pattern\Calculation1.
x.txt");
        Tx.Write(str);
        Tx.Close();
    }
    void get_data1(string str)
    {
        StreamWriter Ty;
        Ty =
File.CreateText(@"D:\EMTI_ARCH_results\Loop_Variable_Pattern\Calculation1.
y.txt");
        Ty.Write(str);
        Ty.Close();
    }
}
```

Figure 4.5 The output from the Instruction_Instrument module, the IWSM (con.)

Whenever the IWSM is running, the embedded instruction will record the start time, the return time and the *iv* values of the loop in text files. A separate example of the structure of the web service for creating the text files is shown in Figure 4.6. Each created text file will be described in the next section.



```
[WebMethod]
public string Calculation1 (double x, double y)
{
while ((x != 3) && (y>0))
        {
            x = (x * x + 2) % y;
        }
return "No Infinite";
}
```

<service_name> text file is *Calculation1.txt*
Input parameters are x and y which these values will be recorded at Calculation1.txt

<service_name_LV> text files are *Calculation1_x.txt* and *Calculation1_y.txt*.
These text files contain values of each iteration variable of the loop.

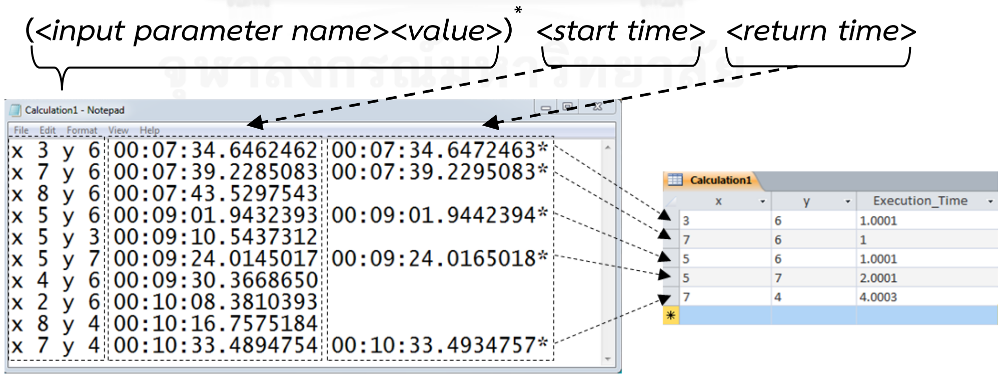
Figure 4.6 The text files created from the web service module structure

4.1.1.1.1 <service_name> text file

This text file will be created from the name of the web service module. This text file is responsible for recording input parameter variables and values, the start time and the return time of the service. The structure of this text file is shown as follows.

$$\boxed{(\textit{<input parameter name><value>})^{*}\ \textit{<start time> <return time>}}$$

The example of the *<service_name>* text file from the structure of checking_Loop1 method is shown in Figure 4.7.


(<input parameter name><value>)* <start time> <return time>
Calculation1 - Notepad
File Edit Format View Help

```
x 3 y 6 00:07:34.6462462 00:07:34.6472463*
x 7 y 6 00:07:39.2285083 00:07:39.2295083*
x 8 y 6 00:07:43.5297543
x 5 y 6 00:09:01.9432393 00:09:01.9442394*
x 5 y 3 00:09:10.5437312
x 5 y 7 00:09:24.0145017 00:09:24.0165018*
x 4 y 6 00:09:30.3668650
x 2 y 6 00:10:08.3810393
x 8 y 4 00:10:16.7575184
x 7 y 4 00:10:33.4894754 00:10:33.4934757*
```

Calculation1

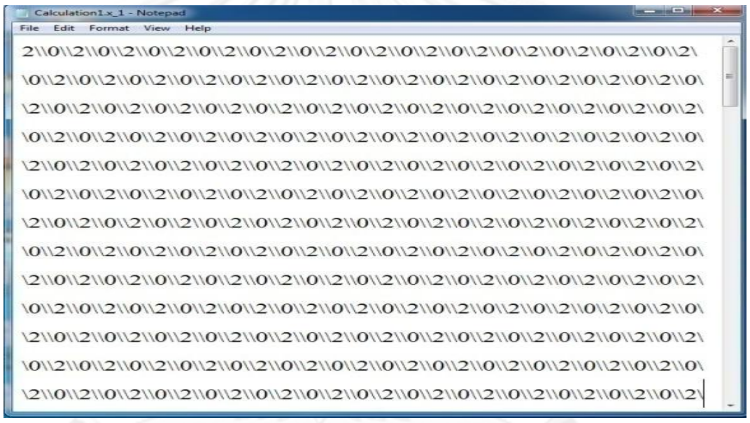
x	y	Execution_Time
3	6	1.0001
7	6	1
5	6	1.0001
5	7	2.0001
7	4	4.0003

Figure 4.7 An example of data in the *<service_name>* text file

According to Figure 4.7, all data are recorded in the text file. Each line will be classified into four data groups: input parameter variables, input parameter values, the start time and the return time. In the case that a service does not have an infinite-loop situation, then the text file will record all values in the structure. In addition, the "*" symbol will be extended to the last alphabet. On the other hand, if the service has an infinite-loop situation, the input parameter values, including the start time without return time, are stored.

4.1.1.1.*2* <service_name_iv> text file

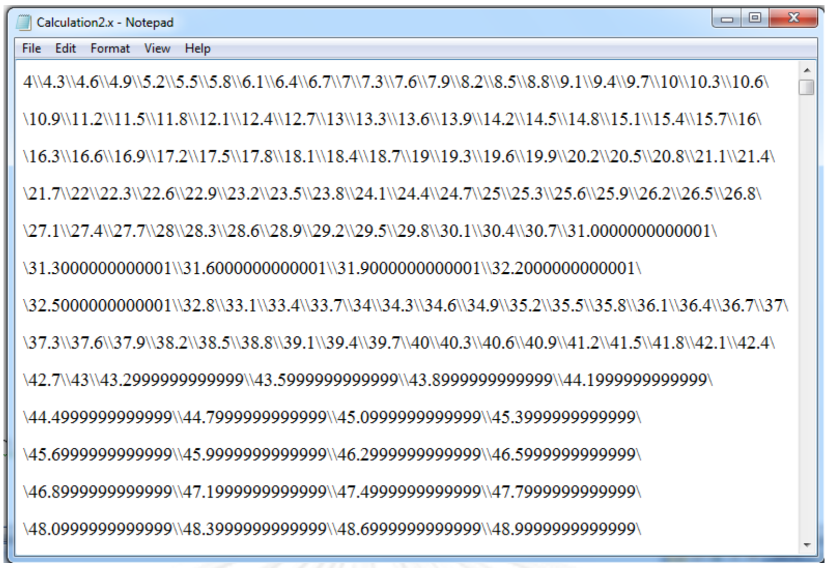
The <*service_name_iv*> text file contains *iv* values. The name of this text file is identified by "service name" and "*iv* name". This text file is created for each *iv* name of the loop condition. Whenever the *iv* values are stored, the separation symbol, "\\", will be used. Examples of <*service_name_iv*> text files are shown in Figure 4.8 – 4.9.



```
2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\
\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\
\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\
\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\
\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\
\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\
\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\
\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\
\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\
\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\
\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\
\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\
\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\\0\\2\
```

Figure 4.8 An example of a text file with a repeating pattern

In Figure 4.8, the *iv* values are stored with two values, 2 and 0. This text file is created from a web service, named Calculation1, and the *iv* name is *x*, Table 3.4 in Chapter 3. Calculation1 has created two text files of *iv* values: *x* and *y*; however, this figure shows only the text file of *x*. The repeated values, 2 and 0, are obtained from changing the *x* value within the loop cycle when *x* starts with 2 and *y* starts with 6.



```
4\\4.3\\4.6\\4.9\\5.2\\5.5\\5.8\\6.1\\6.4\\6.7\\7\\7.3\\7.6\\7.9\\8.2\\8.5\\8.8\\9.1\\9.4\\9.7\\10\\10.3\\10.6\
\10.9\\11.2\\11.5\\11.8\\12.1\\12.4\\12.7\\13\\13.3\\13.6\\13.9\\14.2\\14.5\\14.8\\15.1\\15.4\\15.7\\16\
\16.3\\16.6\\16.9\\17.2\\17.5\\17.8\\18.1\\18.4\\18.7\\19\\19.3\\19.6\\19.9\\20.2\\20.5\\20.8\\21.1\\21.4\
\21.7\\22\\22.3\\22.6\\22.9\\23.2\\23.5\\23.8\\24.1\\24.4\\24.7\\25\\25.3\\25.6\\25.9\\26.2\\26.5\\26.8\
\27.1\\27.4\\27.7\\28\\28.3\\28.6\\28.9\\29.2\\29.5\\29.8\\30.1\\30.4\\30.7\\31.0000000000001\
\31.3000000000001\\31.6000000000001\\31.9000000000001\\32.2000000000001\
\32.5000000000001\\32.8\\33.1\\33.4\\33.7\\34\\34.3\\34.6\\34.9\\35.2\\35.5\\35.8\\36.1\\36.4\\36.7\\37\
\37.3\\37.6\\37.9\\38.2\\38.5\\38.8\\39.1\\39.4\\39.7\\40\\40.3\\40.6\\40.9\\41.2\\41.5\\41.8\\42.1\\42.4\
\42.7\\43\\43.2999999999999\\43.5999999999999\\43.8999999999999\\44.1999999999999\
\44.4999999999999\\44.7999999999999\\45.0999999999999\\45.3999999999999\
\45.6999999999999\\45.9999999999999\\46.2999999999999\\46.5999999999999\
\46.8999999999999\\47.1999999999999\\47.4999999999999\\47.7999999999999\
\48.0999999999999\\48.3999999999999\\48.6999999999999\\48.9999999999999\
```

Figure 4.9 An example of a text file with an increasing pattern

In Figure 4.9, the text file is created from a web service, named Calculation2, and the *iv* name is *x*, Table 3.4 in Chapter 3. Calculation2 creates one text file of *iv* values, which is *x*. The increasing values of x are obtained from changing the *x* value within a loop cycle when *x* starts with 4.

4.1.2 Training Package

After the IWSM has been created from the original web service, the training package will be used by the S/W tester. This package contains three modules: the Selection_Calculation module, the Time_Checking module, and the Pattern_Checking module.

4.1.2.1 Selection_Calculation module

This module is responsible for calculating the EMTI values from the execution time.

4.1.2.2 Time_checking module

The initial upper bound of EMTI values is set as the maximum value of the defining boundary. If the execution time is more than the boundary value, the *iv* values will be checked for patterns. Whenever, the execution time is higher than the boundary value, the Pattern_Checking module is called. Details of each module are described as follows.

4.1.2.3 Pattern_Checking module

Whenever the IWSM runs, the *iv* values of the loop is recorded in the <service_name_iv>text file. The *iv* values are a number of digits separated with “\\” symbols. This module is responsible for finding patterns in the *iv* values. The patterns indicate the running state that cannot reach the final state. Based on Definitions 4 – 8 in Chapter 3, patterns can be classified into two cases as follows:

Case 1: the patterns occur from a repeating set of the *iv* values. The checking technique will be described in Figure 4.10(a) – 4.10(c). Moreover, the algorithm for Case 1 is shown in Figure 4.11.

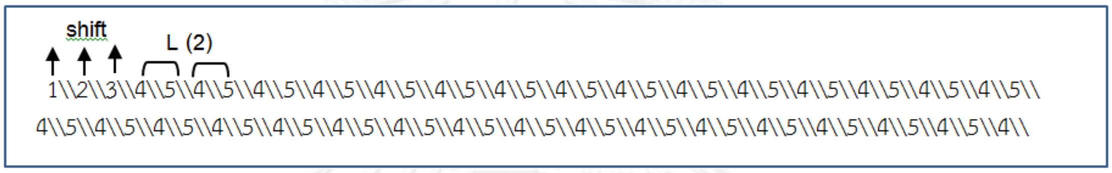


Figure 4.10 (a) Shift the index whenever the value doesn’t have a repetition

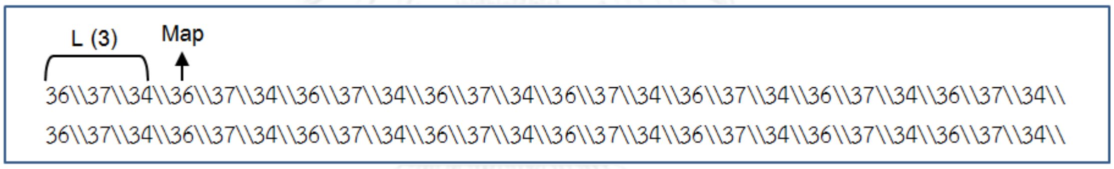


Figure 4.10 (b) The pattern can be found in the first index

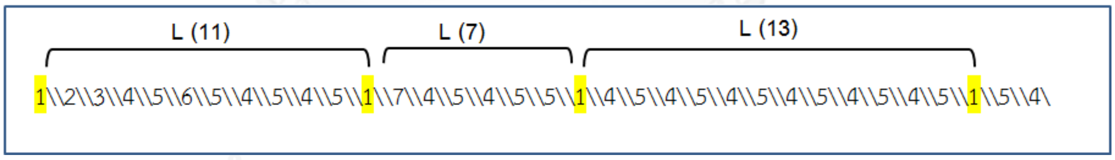


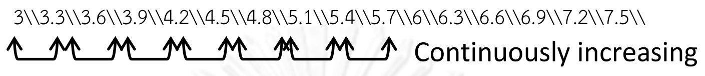
Figure 4.10 (c) The pattern cannot be detected because the range is not of the same size

```
/*      N : a number of iv values
        str : Array of iv values
        idx : Array of indices of iv values
        range : Array of length of each pattern
        size_range : size of the length
        c_range : a number of the same range
        c : count a number of the same iv values
        st1 : status of comparing the same range
        rep_pattern : status of founding the pattern        */
j := 0
for i := 0 to N  //checking index of the same data
  while (j < N)
     if (check index of str[i] by starting at position i) > j
       idx[c] := record index of the same str[i]
        j := idx[c] + 1  //move index for checking
       c++
     else
       j++ //add the index when are not the same data in the text
       loop continue
     end if
  end while
if (c > 1) //if a number of index of the same data more than one
  for i := 0 to c - 2
    range[i] = idx[i + 1] - idx[i] //checking range between the same data
  end for
  for i := 0 to c - 2  //Compare range of each part of the same data
    if (check the same range : range[i] == range[i + 1])
        c_range := c_range + 1;
    end if
  end for
end if
if (c_range >= (c-1))
   st1:= 1//keep status when the same range
end if
if (st1:= 1)
 while(count1 <idx - 1)
     while (count2 < range))
        if (str[(idx[count1] + count2)]:= str[(idx[count1 + 1] + count2)])
       //compare character each part of the range
               count3++
        end if
        if (count3 := range)
               //if a number of the same data equals to the range
           count4++ //count a number of the same data per the range
       end if
            count2++
     end while
        count2 := 0
        count3 := 0
        count1++
 end while
end if
if (count4 := c_range)
        rep_pattern := true
end if
end for
```

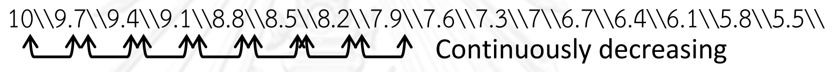
Figure 4.11 Algorithm for detecting the pattern of Case 1

Case 2: the pattern occurring from the situation in which the *iv* has continuously increasing or decreasing values. The checking technique will be described as follows:

a) Type 1: the *iv* value is continuously increasing. Each value will be compared with the next *iv* value for checking the increasing sequence. The hundreds percent of increasing that is used for the conclusion of this pattern. An example of this type is shown as below.



b) Type 2: the *iv* value is continuously decreasing. Each *iv* value will be compared with the next *iv* value for checking the decreasing sequence. The hundreds percent of decreasing that is used for the conclusion of this pattern. An example of this type is shown as below.

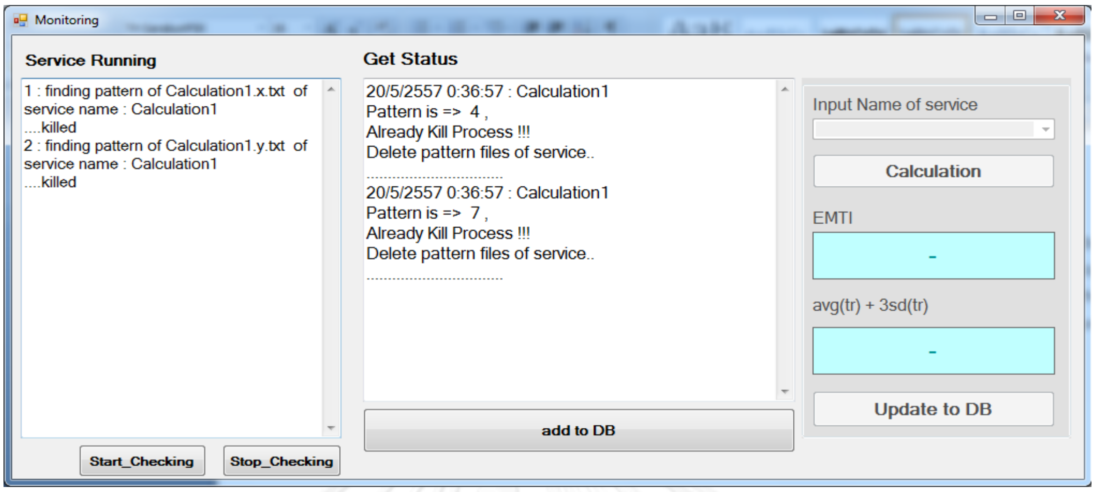


The algorithm for Case2 with both Type1 and Type2 are shown in Figure 4.12.

```
/*     N : a number of iv values
       str : Array of iv values
       increase : a number of rank of increasing
       decrease : a number of rank of decreasing
*/

for i := 0 to N
  if (str[i] < str[i+1])
    increase++;
  else if (str[i] > str[i+1])
    decrease++;
  end if

  if (increase >= N)
    increase_pattern := true
  else if (decrease >= N)
    decrease_pattern := true
  end if
end for
```

Figure 4.12 Algorithm for detecting the pattern of Case 2

Example results from the Time_Checking module and the Pattern_Checking module are shown in Figure 4.13.



```
1 : finding pattern of Calculation1.x.txt  of
service name : Calculation1
....killed
2 : finding pattern of Calculation1.y.txt  of
service name : Calculation1
....killed
```

```
20/5/2557 0:36:57 : Calculation1
Pattern is =>  4 ,
Already Kill Process !!!
Delete pattern files of service..
...............................
20/5/2557 0:36:57 : Calculation1
Pattern is =>  7 ,
Already Kill Process !!!
Delete pattern files of service..
...............................
```

Figure 4.13 Example results from the Time_Checking module and the Pattern_Checking module

4.1.3 Verification Package

This package is used by the user, in which there are three modules, the same as the training package.

4.1.4 Termination Package

This package is called for terminating the IWSM when the infinite-loop situation occurs.

4.1.5 Web Service Repository

The web service repository is for storage of the IWSMs.

4.1.6 Database Design

The database, EMTI_DB, contains three tables: the Loop_Instruction table, the EMTI table, and the <*service_name*> table, as shown in Table 4.1.

Table 4.1 Details of the EMTI_DB database

Table No.	Name of Table	Description
1	Loop_Instruction	The table stores the information of loop structure in the web service module
2	*<service_name>*	The table stores the input parameters and execution time from a set of deterministic input
3	EMTI	The table stores the information on boundary of execution time for checking execution time

The details of each table are explained below.

4.1.6.1 Loop_Instruction Table

After passing the Instruction_Instrument module, the structure of the loop instruction from the web service module will be recorded in the Loop_Instruction table. The fields of the Loop_Instruction table are shown in Table 4.2.

Table 4.2 Structure of the Loop_Instruction Table

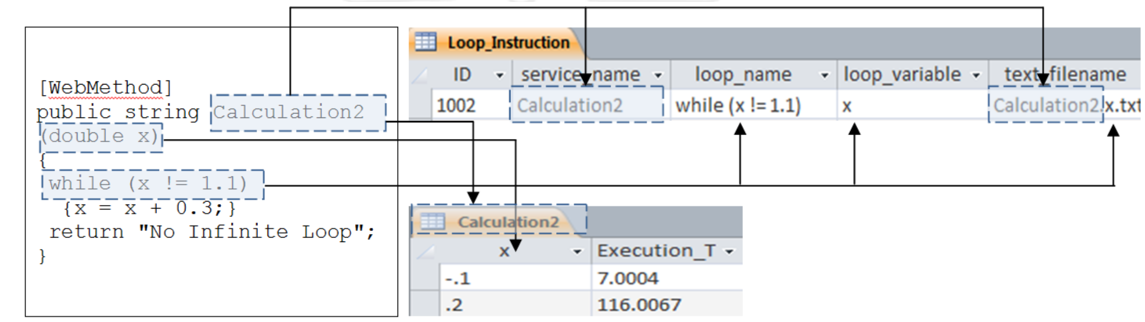
Field Name	Data Types	Description	Extra
Id	Text (4)	Number of record	Primary Key, Not null
service_name	Text (50)	Name of the web service module	Not null
loop_name	Text (150)	Loop structure in the web service module	Not null
Loop_variable	Text (50)	*iv* names of each loop structure	Not null
Text_filename	Text (50)	Name of text file for recording *iv* values	Not null

4.1.6.2 <service_name> Table

<service_*name*> table is created for recording the values of the input parameters. The fields of this table are not stable since it is dependent on the structure of the web service module. The table fields consist of input parameter names and the execution time, as shown in Table 4.3.

Table 4.3 Structure of the <*service_name*>Table

Field Name	Data Types	Description	Extra
<*service input parameters* >*	Text(50)	The value of this field depends on the number of the service input parameters.	Primary Key, Not null
Execution_Time	Text (100)	Each execution time of deterministic initial *iv* values	Not null



```
[WebMethod]
public string Calculation2
(double x)
{
while (x != 1.1)
  {x = x + 0.3;}
 return "No Infinite Loop";
}
```

Loop_Instruction

ID	service_name	loop_name	loop_variable	text_filename
1002	Calculation2	while (x != 1.1)	x	Calculation2x.txt

Calculation2

x	Execution_T
-.1	7.0004
.2	116.0067

Figure 4.14 Extraction data from the original web service module

An example of extracted data from a web service module is shown in Figure 4.14. The web service module, named Calculation2, instrumented its structure to the Loop_Instruction table with service_name, loop_name, loop_variable, and text_filename of the loop. Moreover, the Calculation2 table is created in the EMTI_DB from the identified structure of the <*service_name*> Table, and it has two fields: *x* (the input parameter of the service) and Execution_Time.

4.1.6.3 EMTI Table

The values in the<*service_name*> table are used to calculate the EMTI values, and are recorded to the EMTI table. The EMTI value was calculated using 95% confidence level from the EMT of the deterministic input parameter test cases. The structure of EMTI table is shown in Table 4.4.

Table 4.4 Structure of the EMTI Table

Field Name	Data Types	Description	Extra
Service_name	Text (50)	Name of the service.	Primary Key, Not null
EMTI	Text (255)	Lower bound and Upper bound of the confidence interval of execution mean time.	Not null
max_exeTime	Number (Double)	Upper bound of EMTI value.	Not null

4.2 Experimental Results

4.2.1 Performance of the System

The performance of the proposed architecture is measured based on CPU usage. Figure 4.15(a) and Figure 4.15(b) show the CPU usage of web service algorithms no.1 and no.4 from Table 3.4 in Chapter 3. The solid line represents percentage of the CPU usage under the infinite-loop situation being run by the proposed architecture. The dotted line shows percentage of the CPU usage under the infinite-loop situation based on the original architecture. Each graph shows that whenever the infinite loop occurs and is detected by the proposed architecture, the CPU usage does not trend to increase and to drop as a result of the service termination.

Based on the nine algorithms in Chapter 3, these algorithms were created as the web service modules and run on the IIS web server. The web server is run on a core-i3 CPU, 2.83 GHz with 2 GB RAM. From the results of running these modules, it is clear that the CPU usage is high under the infinite-loop situation. Nonetheless, after detecting the unwanted criteria, these abnormalities are terminated and the CPU usage will be reduced approximately 20%.

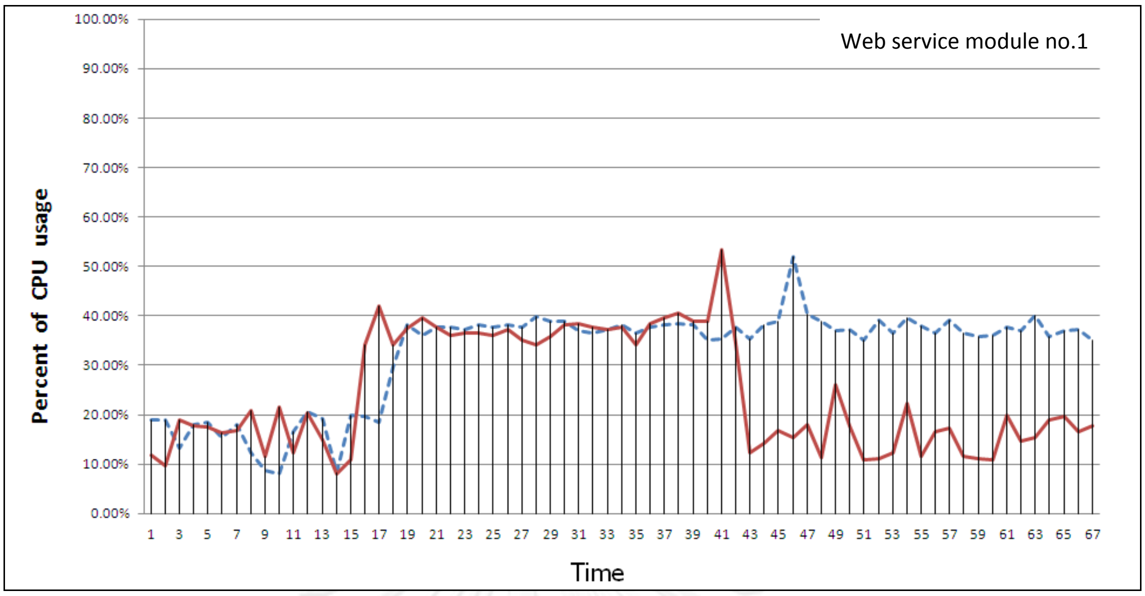


Figure 4.15 (a) Percent of CPU usage of the web service module no.1

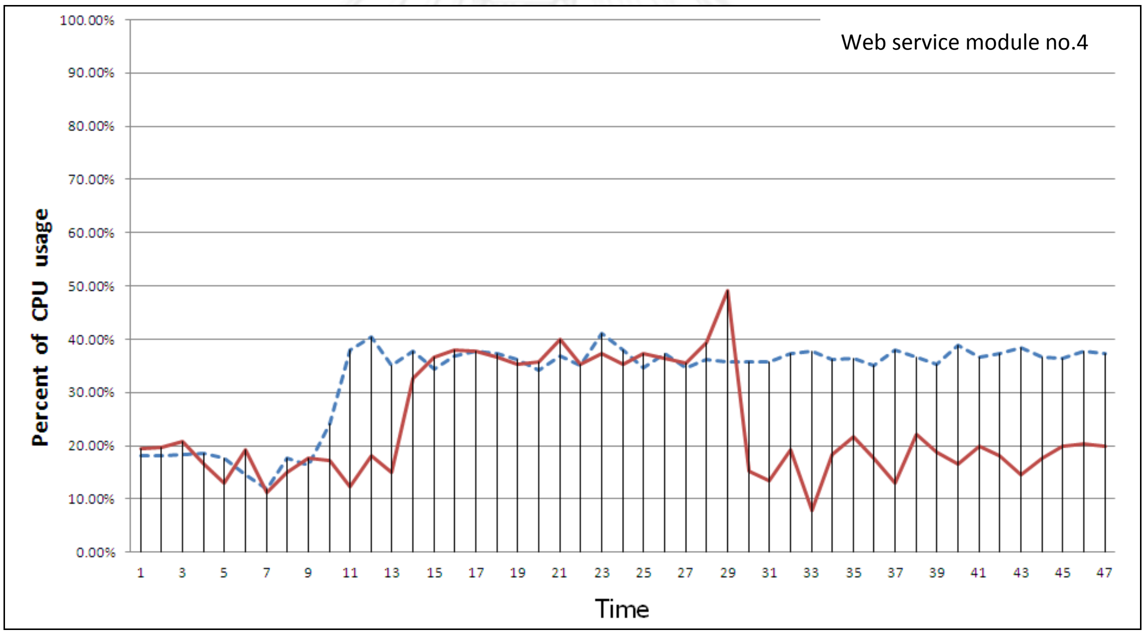


Figure 4.15 (b) Percent of CPU usage of the web service module no.4

Moreover, the experiment was expanded to call sub-function services, which will be described in the next section.

4.2.2 The Main Web Service Module under the Sub-function Services

This evaluation is performed using the architecture of the sub-function services. The web service modules under the sub-functions are designed as shown in

Table 4.5 using five sub-function structure types: sequence, parallel, loop, selection, and composition. The sub-function services are designed on the assumption of the infinite-loop situation, using algorithms in Table 3.4 in Chapter 3. Each main service implements one level of the sub-function type.

Table 4.5 Main web service modules with sub-function services

Sub-function types	Main web service algorithms

Sequence	[WebMethod]
	public string sub_Sequence(double x, double y)
	{
	Cal1.Service c1 = new Cal1.Service();
	c1.Calculation1(x, y);
	Cal2.Service c2 = new Cal2.Service();
	c2.Calculation2(x);
	Cal3.Service c3 = new Cal3.Service();
	c3.Calculation3(x);
	return "return sequence";
	}

Selection	[WebMethod]
	public string sub_Choice(double x, double y, double n)
	{
	if (n > 0)
	{ Cal1.Service c1 = new Cal1.Service();
	c1.Calculation1(x, y); }
	else
	{ Cal2.Service c2 = new Cal2.Service();
	c2.Calculation2(x); }
	return "return Choice";
	}

Table 4.5 Main web service modules with sub-function services (con.)

Sub-function types	Main web service algorithms

Parallel	`[WebMethod]`
	`public string sub_Parallel(double x, double y)`
	`{`
	` Cal1.Service c1 = new Cal1.Service();`
	` Cal2.Service c2 = new Cal2.Service();`
	` Cal3.Service c3 = new Cal3.Service();`
	`  Thread tc1 = new Thread(c1.Calculation1(x, y));`
	`  Thread tc2 = new Thread(c2.Calculation2(x));`
	`  Thread tc3 = new Thread(c3.Calculation3(x));`
	` return "return Parallel";`
	`}`

Loop	`[WebMethod]`
	`public string sub_Loop(double x, double y,double n)`
	`{`
	`  Cal1.Service c1 = new Cal1.Service();`
	`  Cal2.Service c2 = new Cal2.Service();`
	`  Cal3.Service c3 = new Cal3.Service();`
	`  for (int i = 0; i < n; i++)`
	`  {    c1.Calculation1(x, y);`
	`       c2.Calculation1(x);`
	`       c3.Calculation1(x);   }`
	`  return "return Loop";`
	`}`

Composition	`[WebMethod]`
	`public string sub_Compound(double x, double y, double n)`
	`{`
	`        Cal1.Service c1 = new Cal1.Service();`
	`        Cal2.Service c2 = new Cal2.Service();`
	`        Cal3.Service c3 = new Cal3.Service();`
	`        if (n > 0)`
	`        {   c1.Calculation1(x, y);`
	`            c2.Calculation2(x);      }`
	`        else`
	`        {     for (int i = 0; i < n; i++)     c3.Calculation1(x);   }`
	`        return "return Compound";`
	`}`

The CPU usage of the main services is measured in the same way as the first experiment. When the architecture detects the infinite-loop situation from the sub-functions, the sub-functions will be terminated. From the experiment results, when a sub-function is terminated, the main process is also terminated.

Next, proof is drawn for confirming that whenever the infinite-loop situation occurs in a sub-function, the main service will be affected.

Lemma 1: If a sub-function has an infinite-loop situation, then the main service also has an infinite-loop situation.

Proof. Let P, Q be Turing Machines (TMs). Suppose that P calls Q as a sub-TM for performing some specific task. Formally, there exists a set of states in P such that it writes the input to Q’s tape, then P transfers the configuration to the start state of Q. Moreover, Q has a set of states that writes the output back to P’s input tape and transfers the configuration back to P.

This proof can be performed by contradiction. Suppose, for the sake of contradiction, that P halts although Q does not reach the state that transfers the execution back to P. The universal-multi-tape TM, R can be constructed, such that it takes P, Q, and some input string. Then, R writes the input to P’s tape and starts the execution of P. When P writes the input on Q’s tape, R also copy the content on Q’s tape to R’s blank tape. At P’s halting state, R starts the blank tape configuration from P’s halting state to Q’s start state, using the copy input. Thus, R can be viewed as follows.

```
R( <P,Q>, x)
{
        Construct P’ from P in a way such that P’ copy the
Q’s input to its own tape (denoted as x’).
        Execute P’(x)
        Execute Q (x’)
}
```

It is clearly seen that P’ halts, according to the assumption on sub TM Q and the halting assumption we made before. Consequently, the configuration of R must reach the start state of Q on input x’. However, R cannot reach its halt state since Q(x’) cannot transfer the execution back to R. However, from the halting assumption, R must halt. Thus, the contradiction appears.

4.2.3 Reliability of the Web Service

Normally, reliability is measured by the number of failures during program execution. The algorithms in these cases are in critical that the reliability of program can be estimated from the number of infinite-loops occurring.

On the other hand, the execution time of the service can be measured in a period of time which the EMTI related on identify boundary of execution time. For this proposal, the Standard Deviation (SD) is used for the calculation of the EMTI value. Therefore, if the SD has a high value, the IWSM may get low reliability. Normally, a number of test cases are an important for reliability. If a number of test cases are high then the reliability is high as well.

Using only the SD of the execution time for estimation of the service reliability is not enough because there are many factors for estimation, such as the number of iterations of the loop, the number of lines of code, and the number of called sub-functions.

The reliability of web service is evaluated from the probability of infinite-loop situation not occurring. The architecture can detect and stop the process when an infinite loop occurs, thus, the probability of the infinite-loop situation not occurring us equal to one. In contrast, the probability of the infinite-loop situation occurring is equal to zero.

CHAPTER 5
DISCUSSION AND CONCLUSION

5.1 Discussion

Since the infinite loop can cause critical damage to organizations, a warning message must be sent to users before this problem occurs. This research proposes a mechanism with architecture to create a reliable system for detecting and controlling the infinite loop problem. Compared to other protection methods, the verification and validation (V&V) will be performed during the development process, while the non-deterministic loop still has a chance to cause errors during run-time. Therefore, the existing V&V methods cannot completely guarantee that the critical damage will not arise after software delivery.

Considering on the run-time protection methods proposed by various researchers, the input dataset is considered to be the most important factor of the protection mechanism in the methods, since many researchers believe that the run-time error relies on unpredicted values, such as the method proposed in [13]. In addition, the input datasets that have been tested during the software testing phase may or may not cover all unpredictable cases. Thus, some run-time errors have a possibility of occurring, leading to a critical problem, especially when the error is related to loop control.

Since the DIDUCE Model [13] created useful concepts for program testing and helping in evolving a program correctly, it is similar to this research. So, a comparison of the similarities between the DIDUCE Model and the proposed architecture is presented in Table 5.1.

Table 5.1 Comparing techniques between the DIDUCE Model and the proposed architecture

Factors	DIDUCE Model	Proposed architecture
Inputs	Create debugging programs on some inputs. - Analysis on differences of behavior between success and failure at run-time. - Firstly, extracting invariants input from known test cases and checking for invariant violations on the failing cases.	Firstly, analyzing inputs using data training set for which the correct outputs are known. - The *iv* values can be chosen as inputs to the loop. - Consideration state of loop condition that the first, output of loop condition is TRUE. Some of the loop cycle leads to the output of loop condition is FALSE.
long-running programs	Create debugging failures in long-running programs. - To suspect variables or code segments by adding debugging statements, assertions, and breakpoints into the program. - Monitoring all the variables in the program which is better suited than to locate such errors.	Detecting long running time from infinite-loop situation. - Instrumenting instructions into the loop body. - Monitoring *iv* values on the pattern occurrence of the infinite-loop situation.

Similar to the proposed method of this thesis, [6,10-11] have proposed detection techniques using time constraints for program termination. These techniques focus on events and program states within a time interval. However, the time intervals defined by these techniques are static values that cannot be altered even though the situation of program is changed. So, the proposed technique in this research is more realistic, as the time interval is dynamic based on the changing situation and environment. Table 5.2 shows timing techniques of the above mentioned research.

Table 5.2 Research on timing techniques

Other timing techniques	Our timing technique
- Setting timing properties of real-time systems. The time was assigned a time value to event occurrence that is a timestamp for defining max-time [6]. - Monitoring timing constraints in real-time systems with designing bounded violation detection latency and minimum monitoring overhead [10]. - Using timing constraints for analysis of correct and/or erroneous states [11].	- Defining maximum execution time for checking the probability of the infinite-loop situation occurring from current execution time of the process. - The boundary of times will be updated during use.

Not only is the time constraint applied, but the VAF-checking technique presented by [16] is also applied as well. This VAF-checking is a complete loop-checking mechanism. Nevertheless, the VAF-checking technique is much more complicated when compared with the proposed mechanism. This is because the VAF-checking technique will consider the whole body structure of the program before its implementation, while the proposed mechanism in this thesis is free from such cases. Therefore, the proposed mechanism can be easily applied to any type of programming structure with loop instructions.

Another technique called the Jolt system [19] is also similar to the proposed mechanism, since it uses embedded code to control the infinite-loop situation. Therefore, Table 5.3 shows a comparison between the Jolt system and the proposed architecture. According to Table 5.3, the Jolt system might not be able to guarantee on state of occurring the infinite loop. In contrast, the proposed architecture uses two techniques for detecting the infinite-loop situation: timing technique and pattern checking. Therefore, the proposed architecture has more reliability of detecting infinite-loop situations than the Jolt system. Lastly, for controlling the infinite-loop situation, the Jolt system used escaping the loop whenever an infinite loop occurred and continuing the process in the main structure. This might be much appropriate than termination of the entire program. However, the consequence of this action is that it cannot be confirmed that the following execution is correct as according to user's expectations.

Table 5.3 Comparison between the Jolt system and the proposed architecture

Techniques	Jolt system	Proposed architecture
Instrumenting source code into the loop body	✓	✓
Detecting infinite-loop situation	Check state change	Checking boundary of execution time and infinite loop pattern occurring
Controlling infinite-loop situation	Escaping the loop whenever infinite loop occurs	Terminating the program whenever infinite loop occurs

5.2 Conclusion

This research aims to control the unlimited execution time of the process in the infinite-loop situation. The proposed architecture has covered the research objectives.

In the first research objective, there are two main solutions to detect and control the infinite-loop situation. Firstly, the boundary of execution times, the EMTI, was defined for analyzing the infinite-loop situation. This value of EMTI was used for detecting the occurrence of infinite loop. After that, the pattern of *iv* values will be detected when the current execution time is higher than the boundary of execution time. If the pattern of *iv* values can be found, the process will be terminated and reported to the requester.

In the second research objective, the architecture was designed on a web server for detecting and controlling infinite loops of web services. The main architecture was shown with four packages: monitoring package, training package, verifying package, and terminating package that depends on each step of the operation. There are five modules designed in the architecture, consisting of: Instruction_Instrument module, Selection_Calculation module, Time_Checking module, Pattern_Checking module and Termination module. These modules work together for completeness of their tasks. Moreover, the obtained performance of the architecture will be evaluated based on the CPU usage. The reliability of the web-service method will be evaluated by analyzing on the probability of a number of infinite loops occurring after using the architecture.

Additionally, service sub-functions are an important issue that should be considered when the web service is required. This is because the sub-functions of the service can lead to the infinite-loop situation. It can be proven by contradiction that whenever the sub-function has infinite loop, the main service will definitely be affected.

5.3 Limitation

- Firstly, the boundary of execution time was created from a number of data test cases.

- If an infinite loop occurs where a pattern cannot be found, the architecture cannot detect the infinite loop.

- The architecture was implemented using Microsoft Visual C#. Therefore, there are three loop instructions that were used in the C# language for testing the architecture consisting of; while, do…while and for loop.

- In case of the sub-function service was terminated from infinite-loop situation that affects the main service, the main service will be terminated as well.

5.4 Future Work

The proposed architecture cannot automatically classify the loop's types, between deterministic or non-deterministic. From Chapter 3, the architecture can only analyze the structure and variables of the loop. Therefore, in future work, the architecture should be able to automatically classify the types of loop, between deterministic and non-deterministic loops, before instrumenting specific instructions into the loop. Thus, deterministic loop instruction will be unnecessarily instrumented with specific instructions.

Since the termination of the infinite loop in a sub-function causes the termination of the main function, the analysis should be performed to prevent such cases if the result from the sub-function is independent from activities of the main function.

5.5 Extended Work

Although this research focuses on the problem of infinite loops over the distributed system, the study of infinite-loop situations on standalone systems was also performed and presented in Appendix A. Moreover, the web services with sub-functions were also considered. The solution for such problems is proposed in Appendix B where the extended (Web Service Definition Language) WSDL for sub-function services is described.

REFERENCES

1. Linda, R.T. and H.J. Shaw, *Software Metrics and Reliability.* 1999.
2. Thio, N. and S. Karunasekera. *Automatic measurement of a qos metric for web service recommendation*. in *Software Engineering Conference, 2005. Proceedings. 2005 Australian*. 2005. IEEE.
3. IBM_Corporation. *Countable Loops*. [cited 2014 June 18, 2014]; Available from: http://sc.tamu.edu/IBM.Tutorial/docs/CforAIX/CforAIX_html/compiler/concepts/cuppl oop.htm.
4. Robinson, S. *Simulation model verification and validation: increasing the users' confidence*. in *Proceedings of the 29th conference on Winter simulation*. 1997. IEEE Computer Society.
5. Sargent, R.G. *Verification and validation of simulation models*. in *Proceedings of the 37th conference on Winter simulation*. 2005. Winter Simulation Conference.
6. Gergeleit, M., J. Kaiser, and H. Streich, *Checking timing constraints in distributed object-oriented programs.* ACM SIGPLAN OOPS Messenger, 1996. 7(1): p. 51-58.
7. Wasserman, H. and M. Blum, *Software reliability via run-time result-checking.* Journal of the ACM (JACM), 1997. 44(6): p. 826-849.
8. Lee, I., et al., *A monitoring and checking framework for run-time correctness assurance.* 1998.
9. Sugaya, M., et al. *A lightweight anomaly detection system for information appliances*. in *Object/Component/Service-Oriented Real-Time Distributed Computing, 2009. ISORC'09. IEEE International Symposium on*. 2009. IEEE.
10. Mok, A.K. and G. Liu. *Efficient Run-Time Monitoring of Timing Constraints*. in *IEEE Real Time Technology and Applications Symposium*. 1997.
11. Robert, T., J.-C. Fabre, and M. Roy. *On-line monitoring of real time applications for early error detection*. in *Dependable Computing, 2008. PRDC'08. 14th IEEE Pacific Rim International Symposium on*. 2008. IEEE.
12. Meyer, A.R. and D.M. Ritchie. *The complexity of loop programs*. in *Proceedings of the 1967 22nd national conference*. 1967. ACM.
13. Hangal, S. and M.S. Lam. *Tracking down software bugs using automatic anomaly detection*. in *Proceedings of the 24th international conference on Software engineering*. 2002. ACM.
14. de Alba, M.R. and D.R. Kaeli. *Runtime predictability of loops*. in *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*. 2001. IEEE.

15. FARLEX. *The Free Dictionary*. [cited 2013 December 10, 2013]; Available from: http://www.thefreedictionary.com/infinite+loop.

16. Shen, Y.-D., L.-Y. Yuan, and J.-H. You, *Loop checks for logic programs with functions.* Theoretical Computer Science, 2001. 266(1): p. 441-461.

17. Burnim, J., et al. *Looper: Lightweight detection of infinite loops at runtime*. in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*. 2009. IEEE Computer Society.

18. Cheng, B.H., et al., *Software Engineering for Self-Adaptive Systems: A Research Roadmap*, in *Software Engineering for Self-Adaptive Systems*, H.C. Betty, et al., Editors. 2009, Springer-Verlag. p. 1-26.

19. Carbin, M., et al., *Detecting and escaping infinite loops with jolt*, in *Proceedings of the 25th European conference on Object-oriented programming*. 2011, Springer-Verlag: Lancaster, UK. p. 609-633.

20. Lee, C.S., N.D. Jones, and A.M. Ben-Amram, *The size-change principle for program termination.* SIGPLAN Not., 2001. 36(3): p. 81-92.

21. Cook, B., A. Podelski, and A. Rybalchenko. *Termination proofs for systems code*. in *ACM SIGPLAN Notices*. 2006. ACM.

22. Velroyen, H., et al., *Non-termination checking for imperative programs*, in *Proceedings of the 2nd international conference on Tests and proofs*. 2008, Springer-Verlag: Prato, Italy. p. 154-170.

23. Kapur, D. *Automatically Generating Loop Invariants Using Quantifier Elimination–Preliminary Report–*. in *IMACS Intl. Conf. on Applications of Computer Algebra*. 2004. Citeseer.

24. Forouzan, A.B., *Data Communications & Networking (sie)*. 2006: Tata McGraw-Hill Education.

25. McClave, J.T. and T. Sincich, *A First Course in Statistics*. 2011: Pearson Education, Limited.

26. Qi, Z., et al. *FLTL-MC: online high level program analysis for Web services*. in *Services-I, 2009 World Conference on*. 2009. IEEE.

27. Antunes, N. and M. Vieira. *Benchmarking vulnerability detection tools for web services*. in *Web Services (ICWS), 2010 IEEE International Conference on*. 2010. IEEE.

28. Oliveira, R., N. Laranjeiro, and M. Vieira. *A Composed Approach for Automatic Classification of Web Services Robustness*. in *Services Computing (SCC), 2011 IEEE International Conference on*. 2011. IEEE.

29. Karray, M., C. Ghedira, and Z. Maamar. *Towards a self-healing approach to sustain web services reliability*. in *Advanced Information Networking and*

Applications (WAINA), 2011 IEEE Workshops of International Conference on. 2011. IEEE.

30. Groce, A., et al., *Error explanation with distance metrics.* Int. J. Softw. Tools Technol. Transf., 2006. 8(3): p. 229-247.
31. Issarny, V., et al. *Coordinated forward error recovery for composite web services*. in *Reliable Distributed Systems, 2003. Proceedings. 22nd International Symposium on*. 2003. IEEE.
32. Liang, D., et al. *Fault tolerant web service*. in *Software Engineering Conference, 2003. Tenth Asia-Pacific*. 2003. IEEE.
33. Liu, L., et al. *A fault-tolerant framework for web services*. in *2009 WRI World Congress on Software Engineering*. 2009.
34. Liu, A., et al., *Facts: A framework for fault-tolerant composition of transactional web services.* Services Computing, IEEE Transactions on, 2010. 3(1): p. 46-59.
35. Fang, C.-L., et al. *A redundant nested invocation suppression mechanism for active replication fault-tolerant web service*. in *e-Technology, e-Commerce and e-Service, 2004. EEE'04. 2004 IEEE International Conference on*. 2004. IEEE.
36. Santos, G.T., L.C. Lung, and C. Montez. *Ftweb: A fault tolerant infrastructure for web services*. in *EDOC Enterprise Computing Conference, 2005 Ninth IEEE International*. 2005. IEEE.

Appendix A
The proposed architecture based on a standalone system

On the standalone system, the proposed architecture includes two main phases consisting of eight modules, as shown in Figure A-1.

The first phase is the testing of the software process; there are three important modules: the checking module (*CM*), the EMTI Module (*EMTIM*), and the Embedded Module (*EM*). The second phase is the deployment phase, or the running phase. This phase has two subsystems: controlling and learning subsystems. The controlling subsystem has three modules: the time checking module, the reporting module, and the termination module. Moreover, the learning subsystem has two modules: the time recording module and the EMTI module.

The testing phase is responsible for automatically tracking the input software. This tracking is performed by instrumenting the time measurement instructions into a part of the non-Deterministic loop within the software. This phase tests the loop processing time based on the calculated EMT and EMTI values from the normal execution times. On the other hand, the running phase checks the processing time of the loop instruction during run-time using the upper bound of the EMTI from the EMTI_DB. Moreover, this phase adjusts the EMT and EMTI values. Details of each phase will be described in the next section.

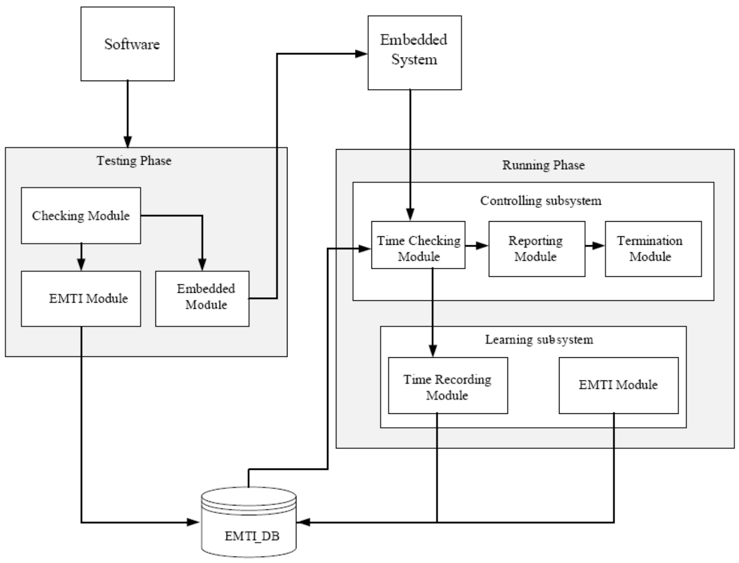


Figure A-1 System Framework

- The testing phase

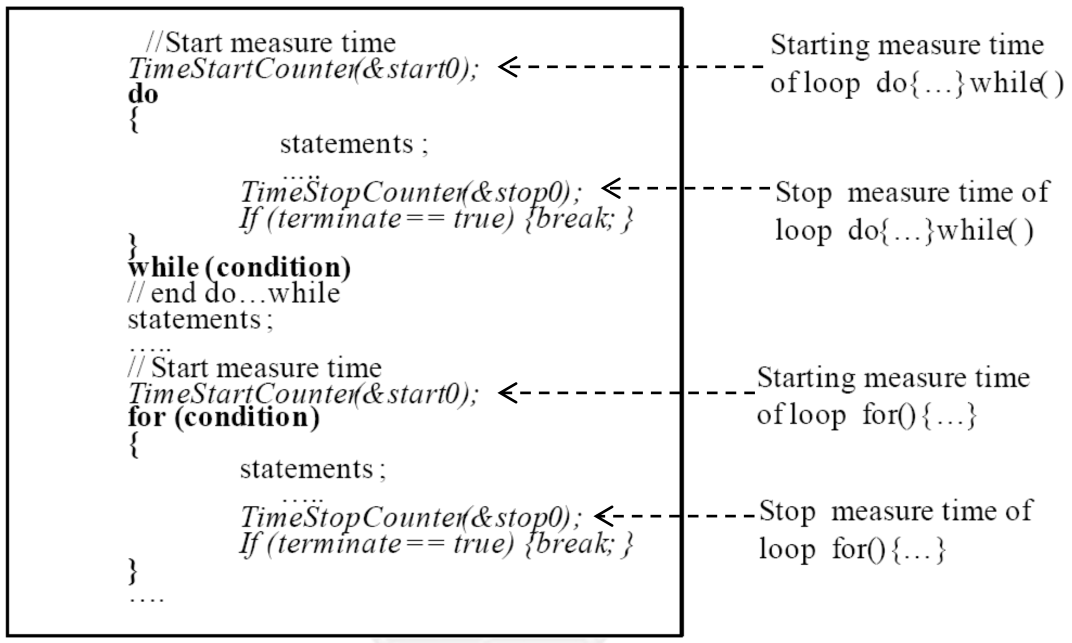
- Checking Module (CM)

In order to prevent the infinite-loop critical problem, the CM checks loop instructions in the software code. Generally, the loop attributes consist of initial values, conditions, and counters. The termination of the loop process is dependent on the loop attributes, which can be classified in two types: static and dynamic attributes. The static attributes of the loop refer to the situation of loop that has finite number to perform its tasks, while the dynamic attribute refers to the situation that the number of loop cycles cannot be determined during the execution. Moreover, the termination of loop is dependent on the value obtained from the input variables of the loop conditions or values of the variables during the loop process which are related to the loop condition at run-time.

Thus, the CM is responsible for checking the existence of the loop instructions, such as $Do \ldots While()$, $While()$, or $For()$ within the software. In this module, the input software has two states under the pre-compile process.

1) *Automatic software tracking*: When input software is entered into this state, each command of this software will be scanned for loop instructions.

2) *Instrument time measurement instruction*: This state will instrument instructions for time measurement into the input software. The measurement of time starts when the loop starts its processes and stops when the loop is terminated, as show in Figure A-2.



```
   //Start measure time
TimeStartCounter(&start0);
do
{
            statements ;
            ....
         TimeStopCounter(&stop0);
         If (terminate== true) {break; }
}
while (condition)
// end do…while
statements;
.....
// Start measure time
TimeStartCounter(&start0);
for (condition)
{
         statements;
            .....
         TimeStopCounter(&stop0);
         If (terminate== true) {break; }
}
....
```

Starting measure time of loop do{…}while( )
Stop measure time of loop do{…}while( )
Starting measure time of loop for(){…}
Stop measure time of loop for(){…}

Figure A-2 Input software with instrumented time capturing instructions

Therefore, after passing this module, the software will be sent to two modules: the EMTI module (*EMTIM*) and the Embedded Module (*EM*). These modules will be described below.

- **EMTI Module (EMTIM)**

The EMTIM is responsible for recording the normal execution times of the software. The EMTIM of the testing subsystem will find an initial EMTI value. The testers generate the input data for software testing. Thus, the input data set of the testing phase must cause an infinite loop. Conditions of the tests are defined below.

- Controlled loops: Controlled loop refers to the situation of all input data sets that will not create any infinite-loop situations.

- Terminated loops: Terminated loops mean the controlled situation under the infinite processing time. The termination of this infinite loop uses the upper bound of the valid time interval of the normal loop execution.
- Relation among of variables of loop conditions: the variables for controlling loop conditions are identified into two types: internal and external variables. The internal variables are all variables existing in the loop, including input commands within the loops. The external variables are all variables that send data to the loop process. Thus, any variables of the loop condition can be defined under the relation between internal and external variables.

The results from the test that are valid execution times will be recorded to the EMTI_DB and will be used for EMT calculation using a 95% confidence level. Consequently, the initial value of the EMTI is declared before the software is delivered.

- **Embedded Module (EM)**

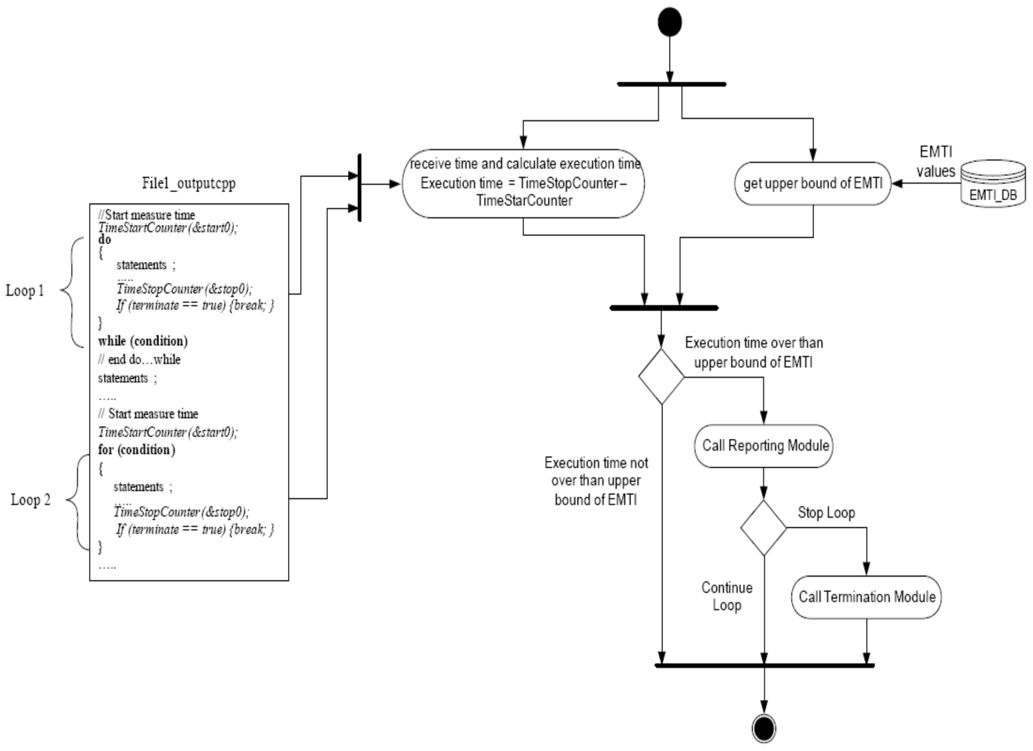
The EM is responsible for embedding the time capturing instructions within the loops of the software. So, this EM will embed only the commands that prevent the unlimited computing time from the loop instruction. Then, the command for starting capture time is added with the TimeStartCounter attribute and the command for stopping capture time is added with the TimeStopCounter attribute. After passing the EM, the software will create a new output file for the run-time process.

- The running phase

After passing the testing process, the software is ready for use with the time capturing instructions from the EM. Under this phase, the software is implemented with two main subsystems: the controlling subsystem and the learning subsystem. The details of each subsystem will be described, as follows:

- Controlling subsystem

The controlling subsystem is responsible for managing the running software when infinite-loop situation occurs. The controlling subsystem consists of three important modules: the time checking module (*TCM*), the reporting module (*RM*), and the termination module (*TM*). The TCM uses the loop starting time and the loop stopping time for calculating the time interval of the loop. Thus, the TCM compares the loop execution time with the upper bound of the EMTI in the EMTI_DB. If the execution time is more than the defined upper bound of the EMTI, then the RM will be called to create a message to inform the user. Moreover, the TM will be called to cancel the software process. The activity diagram of the controlling subsystem is shown in Figure A-3.


File1_outputcpp
Loop 1
Loop 2
receive time and calculate execution time
Execution time = TimeStopCounter – TimeStarCounter
get upper bound of EMTI
EMTI values
EMTI_DB
Execution time over than upper bound of EMTI
Call Reporting Module
Stop Loop
Call Termination Module
Execution time not over than upper bound of EMTI
Continue Loop

```
//Start measure time
TimeStartCounter (&start0);
do
{
     statements ;
     .....
     TimeStopCounter (&stop0);
     If (terminate == true) {break; }
}
while (condition)
// end do…while
statements ;
.....
// Start measure time
TimeStartCounter (&start0);
for (condition)
{
    statements ;
    .....
    TimeStopCounter (&stop0);
     If (terminate == true) {break; }
}
.....
```

Figure A-3 Activity diagram of the controlling subsystem

- Time Checking Module (TCM)

In the first step, each EMTI value is obtained from the testing subsystem and stored in the EMTI_DB that is chosen for checking in this module. The TCM receives the loop starting time and the loop stopping time, and then the loop execution time is calculated. The loop execution time will be compared with the upper bound of the EMTI value.

The comparison result mentioned above indicates whether the executing loop is in the normal or abnormal states. The normal state is the situation that the loop execution time is less than or equal to the upper bound of the EMTI value. On the other hand, the abnormal state is the situation that the execution time has the potential to exceed the upper bound of the EMTI value. Moreover, every loop execution time in the normal state will be recorded in the EMTI_DB for recalculation of the EMTI value in the EMTI module of the learning subsystem.

- Reporting Module (RM)

When the infinite-loop situation is indicated by the TCM, the RM is called for process termination. The RM is responsible for creating a warning message for the user. The message will be displayed to the user as a dialogue. Moreover, the user must send a response to the dialogue message whether to continue or terminate the process. If the user chooses to continue the process, the RM will return all checking processes back to the TCM as the normal process, marked as an abnormal state. Then, the software keeps running whereas every five seconds the warning message will be displayed to the user again. On the other hand, the warning message will disappear when the software finishes executing or the user wants to terminate the process.

- Termination Module (TM)

The TM is responsible for the software process termination according to the result of the TCM and RM. Thus, the TM will be performed whenever the user chooses to terminate the computing process.

- Learning subsystem

- Time Recording Module (TRM)

Previously, the EMTI value of each loop mechanism was set as an initial value from the testing subsystem. Thus, the TRM receives the execution time from the TCM and sends it to the EMTI_DB. The administrator identifies a number of normal states and records them in the EMTI_DB. As a consequence, the recalculating of the

EMTI value will be performed in order to reset the EMTI value in the actual processes. Moreover, the number of occurring infinite-loop situations will be reported to the system administrator for problem consideration.

- **EMTIM**

The EMTIM in the learning subsystem will select the loop execution times of all normal states from the EMTI_DB for recalculation of the EMTI value. The calculation uses values of the remaining unused loop execution times stored in the EMTI_DB

For example, there are 30 remaining unused loop execution times of the normal state; the new EMTI will be calculated based on these remaining values. Meanwhile, the TRM still records the incoming valid values of the other executions of the software.

- The proposed solution under the standalone architecture

The architecture is designed and implemented using Microsoft Visual C++. The software for evaluation used random test cases in seven loop instruction patterns, as follows:

1. The algorithm has only one loop instruction: *for*, *while*, and *do…while*.

2. The algorithm has only one loop instruction and calls a functional instruction where there is no loop instruction existing in the called function.

3. The algorithm has only one loop instruction and calls a functional instruction where loop instructions exist in the called function.

4. The algorithm has more than one loop instruction without nested loop instruction.

5. The algorithm has more than one loop instruction, but it has some nested loop instructions.

6. The algorithm has loop instructions and selection instructions that some works are overlapped between them.

7. The algorithm has loop instructions and selection instruction that there is no work to be overlapped among them.

Based on the seven patterns above, fifteen different software programs were created by a generator program for efficiency measurement. Moreover, there are 100 dissimilar data test sets used as input to the 15 testing software programs. Table A-1 shows various loop conditions in the testing files where each condition can encounter the infinite-loop situation.

Table A-1 The loop instruction case conditions for testing

Loop instruction condition	*Loop exit condition*
for(i = 0; i!= 10; i++)	i =10
while (ch!= ‘c’)	ch =‘c’
while (x != 10)	x = 10
while (a!= b)	a = b
while ((i%2) != 0)	i mod 2 equal 0.
while ((a = b|c) || (b = c|d))	(a = b|c) and (b = c|d) are false
while (x = function())	The value from function() return not equal x

Based on the conditions defined in Table A-1, the test results of the fifteen programs are presented as a line graph in Figure A-4. The graph consists of two lines that represent the two situations of the experiments. The first line represents the infinite-loop situation before using the architecture, while the second line represents the infinite-loop situation after using the architecture. According to the line before using the proposed method, it is clear that within 100 data test sets, every program has a chance to be trapped in the infinite-loop situation. Nevertheless, after the implementation of the proposed solution, the number of infinite-loop situations of software is reduced to zero.

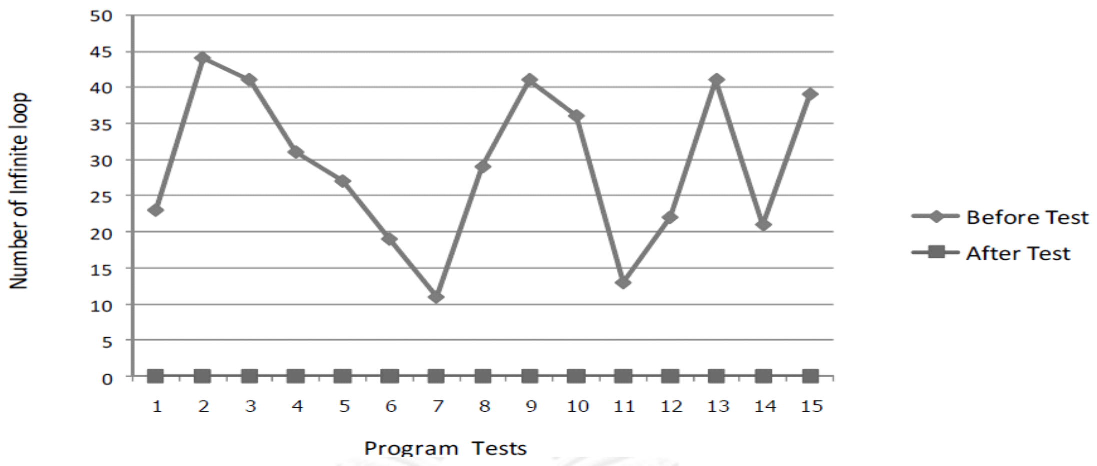


Figure A-4 The graph comparing the number of infinite loops before and after using the system

As shown in the results, the proposed architecture can detect the infinite-loop situation whenever an invalid condition occurs during the run-time process. Moreover, there is no side effect from software termination. Therefore, users can be ensured that their system is secured and trustable.

Appendix B
Extending WSDL for Calling Sub-function

In this case, the web service is separated into two parts: the main service and sub-function services. The main service is the service called from the user or application program. The sub-function service is the external services called from the main services.

In the consideration, if the infinite-loop situation occurs in sub-function services, the infinite-loop situation will occur in the main service as well. Therefore, the web-service method algorithm is a variant of the algorithms for calling sub-functions. In the next section, the exposed sub-function services in the main service algorithm will be expanded by extending the Web Service Definition Language (WSDL).

This section proposes the extended WSDL for sub-function services. The method approach is named as the Extended EMTI (*EEMTI*). The method extends the description of sub-function structures for calling external services in the algorithm. The XML schema design in the WSDL file explains the display of all calling external services with specific flows.

The service algorithm can identify the path of the called sub-function so that the path can expose the structural flow of the called sub-function. The sub-function structural flow in the service algorithm is defined in Definition B-1. Moreover, the service execution time is approximated in each flow type referring to Definition B-2, respectively.

Definition B-1:

The flow types of the called external services can be classified as sequence, parallel, conditional and loop. These types are analyzed from the syntax of the programming language for calling external services.

According to all flow types defined in Definition B-1, the execution time of each flow will rely on different criteria. Firstly, the simplest flow type is the sequence flow. The completeness of the sequence flow is dependent on the completeness of every sequential process of the task.

Secondly, the parallel flow refers to concurrent tasks. The parallel flow is divided into two different alternatives: the minimum time execution, and maximum time execution. The minimum time execution occurs when only one sub-task finishes its process and the other sub-tasks will be ignored. The result of the first finished sub-task will be used in the next process. On the other hand, the maximum time execution refers to the situation that every sub-task must complete their assignments before moving to the next process.

Differing from other flow types, the execution time of the condition flow relies on the tasks under the satisfied condition, while the execution time of the loop is the accumulation of execution times of all cycles. Therefore, the time for each flow can be defined as Definition B-2.

Definition B-2:

Let $T_{P[RTTD][FT]}$be the execution time of dependent agents under the flow type$[FT]$. Let TS_ibe the service time of the external agent i^{th}. The value of $T_{P[RTTD][FT]}$in each process flow type is defined as follows:

$$T_{P[RTTD][Sequence]} += \sum_{i=1}^{n} TS_i$$

$$T_{P[RTTD][Parallel]} += \mathrm{MAX}, \mathrm{MIN}\{TS_i | \mathrm{i} = 1, \ldots, \mathrm{n}\}$$

$$T_{P[RTTD][Condition]} += TS_1 | TS_2 | TS_3 | \ldots | TS_{ni}$$

$$T_{P[RTTD][Loop]} += \sum_{i=1}^{k} \sum_{j=1}^{n} TS_j$$

where	n	is the number of services to be executed within the Web Service Composition (WSC),
	k	is the number of loops to be executed within the WSC.

Moreover, the limitation of the execution time under the infinite-loop problem is defined using the maximum value. Since there are various flow types as previously mentioned, all of these flows can be implemented in the main service algorithm. Thus, if the sub-function is an infinite-loop situation, the infinite-loop situation will also occur in the main service.

Since the general flow types, as defined in Definition B-1 and Definition B-2, can be supported by the WSDL, the sub flow is counted as a new flow type that consists of various general flow types. Therefore, the extended framework of the WSDL must be extended to support such cases.

The Extending WSDL Schema (EWS) is written in XML schema as an extended part of WSDL in the web service agent. This module is embedded in the message type of the WSDL definition for describing the sub-function of the architecture. This extended schema is distributed to other web service agents for updating their WSDL descriptions. The parameters of the referred EWS are shown in Figure B-1.

```
1: <s:element name= [agentName]_“Extend”>
2:   <s:complexType>
3:     {<s:element subType=[type](_type)*/>
4:      <s:binding WSDL_transport= “http://www.xxx.xx”
                    exAgent =[exAgentName]/>}*
5:   </s:complexType>
6: </s:element>
```

Figure B-1 The EWS in the WSDL document message type

According to Figure B-1, all parameters can be described, as below.

- $[\boldsymbol{agentName}]$ **:** This parameter refers to the agent web service’s name.
- $[\boldsymbol{type}]$: This parameter is used to distinguish the structure flow of calling the sub-function between sequence, parallel, choice, and loop.
- $[\boldsymbol{exAgentName}]$: This parameter refers to the name of the external service.

Moreover, these parameters must be defined under tags named “s:element”, “s:complexType” and “s:binding”, as shown in Figure B-1. Each command line in the EWS can be elaborated as follows:

- First line 1:<s:element name = $[\boldsymbol{agentName}]$_“Extend”>. This tag contains the $[\boldsymbol{agentName}]$ parameter, and “_” symbol. Supposing that *Root1* is the name of the web service agent, then the tag of the first line is<s:element name = “*Root1*_Extend”>.

- Second line 2:<*s:complexType*>. This tag states the beginning of the extended structure. Under this specific tag, there are two main tags, <s:element> and <*s:binding*> tags, as written in line number 3 and 4. However, the occurrence of these two tags that can be multiple, as marked by "*", because there can be many sub-functions within the web service algorithm.

Within the <*s:element*>, the *subType* attribute is used to identify the sub-function with the flow types. In the simple sub-function, it is followed by $[\boldsymbol{type}]$ parameter, as shown in Table B-1. Otherwise, it is followed by $(\_\boldsymbol{type})^*$ symbol. The value of "*" symbol referring to the repeated process can be assigned in the range of 0 to *N*. Examples of these sub cases are shown in Table B-1, and Table B-2 respectively.

Table B-1 Basic constraint format of sub type in <s:element subType> tag

Types	Example Tags
Sequence	<s:element subType= "Sequence" />
Parallel	<s:element subType= "Parallel" />
Choice	<s:element subType= "Choice" />
Loop	<s:element subType= "Loop" />

Table B-2 Examples of compound format of sub type in <s:element subType> tag

Types	Example Tags
Loop_Choice	<s:element subType= "Loop_Choice" />
Sequence_Loop_Parallel	<s:element subType= "Sequence_Loop_Parallel" />

Besides the <s:element> tag of <s:complexType>, the <s:binding> tag identifies all elements under the *subType* attribute that must be blinded. Thus, two important attributes must be defined: *WSDL_transport* and *exAgent*. The *WSDL_transport* attribute shows the URL of the external service's WSDL, while the *exAgent* attribute shows the name of the required external service. This tag can occur more than once under each sub type.

Afterwards, the EWS is instrumented in the WSDL file, and then the file will be set upon its original URL of the service.

- Monitoring Sub-function (MNS)

The MNS is a module responsible for tracking the sub-functions defined in the extended WSDL file. First, the main service checks all sub-elements from its WSDL. Therefore, other levels of the sub flow is exposed to details from the WSDL of the main web service by checking the WSDL's URL, stated in the *WSDL_transport* attribute and the external service name in the *exAgentName* attribute. All elements of the sub-function will be verified from the boundary to the central part of the sub-function. Afterward, all elements are recorded in the EMTI_DB at the main service.

- Case study of extended WSDL for calling sub-function

The example of the sub-function of Root1 agent is elaborated using flow chart in Figure B-2. Root1 is the main web service agent with two sub-functions. The flow chart shows the levels of the sub-functions under the Root1 agent algorithm. The sub-function type of Root1 is the first level sub-function, where there are three sub-function types: the Sequences, the Loop, and the Loop_Choice. The solid line shows the flow of algorithms in the first level of Root1 (Root1.Level1).

Within the Sequence of the first level, Level1, there are two external services called, namely A1 and B2. Under the loop of Level1, there is a call for an external service, named C1. Moreover, there is a choice within the loop. This structure is called a Loop_Choice structure. Under the Loop_Choice structure, there are two external services called: D2 and E3. Therefore, the sub elements of Level1 (Root1.Level1) contain five elements: A1, B2, C1, D2, and E3.

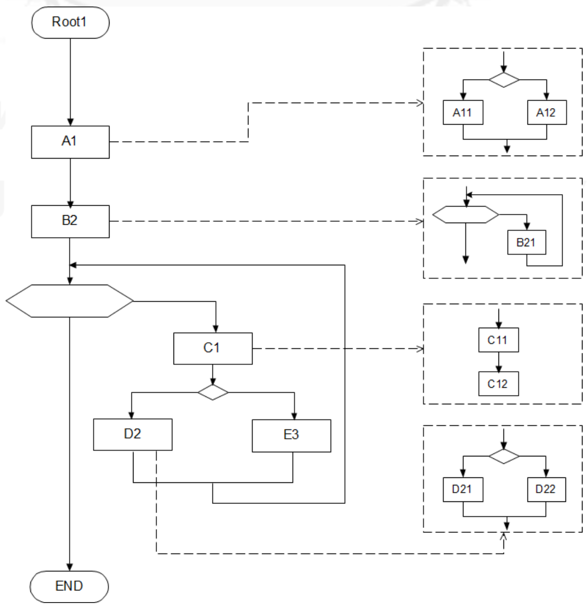


Figure B-2 Example of sub-function flow of "Root1" agent

Referring to the structure described above, the extended description of the Root1 agent has three sub-functions: the Sequence, the Loop, and the Loop_Choice. The extended WSDL description of Root1 agent is shown in Figure B-3.

```
<s:element name="Root1_Extend">
        <s:complexType>
                <s:element name="Sequence"/>
                        <s:binding WSDL_transport="http//xxx.x.xx/x/x" exAgent="A1"/>
                        <s:binding WSDL_transport="http//xxx.x.xx/x/x" exAgent ="B2"/>
                <s:element name="Loop"/>
                        <s:binding WSDL_transport="http// xxx.x.xx/x/x" exAgent ="C1"/>
                <s:element name="Loop_Choice"/>
                        <s:binding WSDL_transport="http// xxx.x.xx/x/x" exAgent ="D2"/>
                        <s:binding WSDL_transport="http// xxx.x.xx/x/x" exAgent ="E3"/>
        </s:complexType>
</s:element>
```

Figure B-3 The Extending WSDL of the "Root1" agent

After tracking all sub elements of Root1.Level1, all sub-functions in other levels are tracked by the MNS. Then, each element of level1 calls an external service in its algorithm, as listed below.

(1) A1 has a Choice, A11 and A12,

(2) B2 has a Loop, B21,

(3) C1 has a Sequence, C11 and C12,

(4) D2 has a Parallel, D21 and D22,

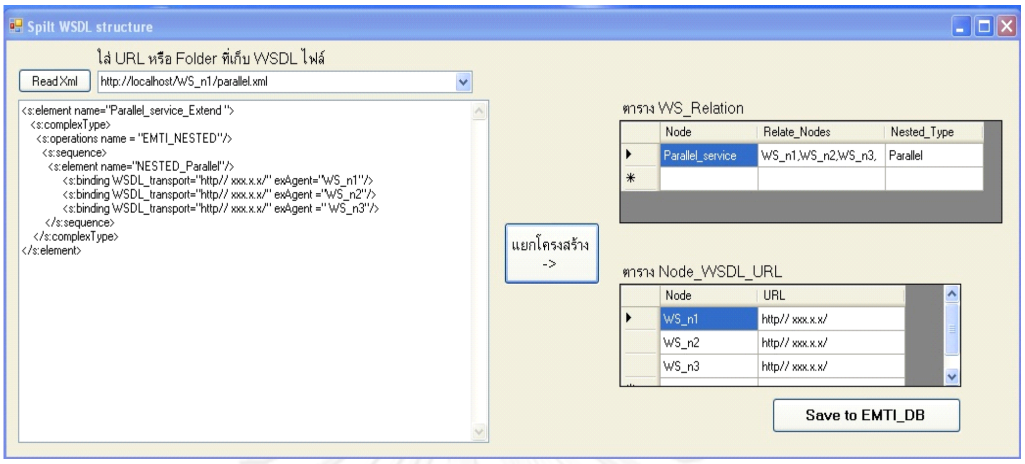
(5) E3 is the closed element.

The details of these sub-functions in the second level, level2, called as Root1.Level2, are shown in the dotted line in Figure B-4.

```
<s:element name="Root1_Extend ">
        <s:complexType>
                <s:element name="Sequence"/>
                        <s:binding WSDL_transport="http//xxx.x.x/" exAgent="A1"/>
                        <s:element name="A1">
                                <s:complexType>
                                        <s:element name="Choice"/>
                                        <s:binding WSDL_transport="http// xxx.x.x/" exAgent ="A11"/>
                                        <s:binding WSDL_transport="http// xxx.x.x/" exAgent ="A12"/>
                                </s:complexType>
                        </s:element>
                        <s:binding WSDL_transport="http//xxx.x.x/" exAgent ="B2"/>
                        <s:element name="B2">
                                <s:complexType>
                                        <s:element name="Loop"/>
                                        <s:binding WSDL_transport="http// xxx.x.x/" exAgent ="B21"/>
                                </s:complexType>
                        </s:element>
                <s:element name="Loop"/>
                        <s:binding WSDL_transport="http//xxx.x.x/" exAgent ="C1"/>
                        <s:element name="C1">
                                <s:complexType>
                                        <s:element name="Sequence"/>
                                        <s:binding WSDL_transport="http// xxx.x.x/" exAgent ="C11"/>
                                        <s:binding WSDL_transport="http// xxx.x.x/" exAgent ="C12"/>
                                </s:complexType>
                        </s:element>
                <s:element name="Loop_Choice"/>
                        <s:binding WSDL_transport="http//xxx.x.x/" exAgent ="D2"/>
                        <s:element name="D2">
                                <s:complexType>
                                        <s:element name="Parallel"/>
                                        <s:binding WSDL_transport="http// xxx.x.x/" exAgent ="D21"/>
                                        <s:binding WSDL_transport="http// xxx.x.x/" exAgent ="D22"/>
                                </s:complexType>
                        </s:element>
                        <s:binding WSDL_transport="http//xxx.x.x/" exAgent ="E3"/>
        </s:complexType>
</s:element>
```

Figure B-4 All sub-functions description of the “Root1” agent

There are seven elements of Root1.Level2: A11, A12, B21, C11, C12, D21, and D22. Assume that every element of Root1.Level2 is the closed element. Thus, the sub-function of the Root1 agent has only two sub levels: Level1, and Level2. The overall sub-function of the Root1 agent after passing the MNS is shown by the WSDL description, in Figure B-4. The application for detecting the structure of the sub-function from the extended WSDL files is shown in Figure B-5.



http://localhost/WS_n1/parallel.xml

```
<s:element name="Parallel_service_Extend ">
  <s:complexType>
   <s:operations name = "EMTI_NESTED"/>
    <s:sequence>
     <s:element name="NESTED_Parallel"/>
         <s:binding WSDL_transport="http// xxx.x.x/" exAgent="WS_n1"/>
         <s:binding WSDL_transport="http// xxx.x.x/" exAgent ="WS_n2"/>
         <s:binding WSDL_transport="http// xxx.x.x/" exAgent =" WS_n3"/>
     </s:sequence>
   </s:complexType>
</s:element>
```

ตาราง WS_Relation

Node	Relate_Nodes	Nested_Type
Parallel_service	WS_n1,WS_n2,WS_n3,	Parallel

ตาราง Node_WSDL_URL

Node	URL
WS_n1	http// xxx.x.x/
WS_n2	http// xxx.x.x/
WS_n3	http// xxx.x.x/

Figure B-5 Application for detecting sub-function from extended WSDL files

From the abovementioned Definition B-1 and Definition B-2, the flow types of calling external services related to the main program on rank of occurring infinite-loop situation.

The extended WSDL is a basic for evaluation of reliability of the main program, such as ranking execution paths. Therefore, each flow type can be approximated with percent of infinite-loop occurring for probability of basic path as follows.

In the sequence type; if some sub-functions have a non-deterministic loop, the percent of calling the sub-functions is 100% because the algorithms have only one path for this type.

In the parallel type; if some sub-functions have a non-deterministic loop, the percent of calling that sub-function is 100% and the quantity of resources used is higher than other types because each sub-function is implemented at the same time.

In the choice type; if some sub-functions have a non-deterministic loop, the percent of calling the sub-functions dependent on the number of conditions and called sub-functions.

In the loop type; nested loops will occur in the algorithm, if some sub-functions have non-deterministic loops. The sub-functions will be implemented under the loop. Thus, whenever an infinite loop occurs in the sub-function, the loop iteration will be stopped to wait for the result from the sub-function. The percent of calling that sub-function depends on the instructions under the loop such as the compound type.

After checking the sub-function, the sub-function will be recorded in the EMTI_DB. There are two tables that are extended to the EMTI_DB: the WS_relation Table and Node_WSDL_URL Table. The sub-function will be exposed with the number of levels and nodes, including relations between nodes. The structures of these tables are shown in Table B-3 and Table B-4.

- **WS_relation Table**

Table B-3 Structure of the WS_relation Table

Field Name	Data Types	Description	Extra
service_name	Text (50)	Name of web-service method	Primary Key, Not null
relate_nodes	Text (255)	Names of sub-function service in each flow type	Primary Key, Not null
structure_type	Text (50)	flow type : sequence, parallel, loop and choice	Not null

- **Node_WSDL_URL Table**

Table B-4 Structure of the Node_WSDL_URL Table

Field Name	Data Types	Description	Extra
node	Text (50)	Name of sub-function service	Primary Key, Not null
URL	Text (255)	URL of WSDL file of the service	Not null

Appendix C
List of Publications

1) Srirajun, N., Bhattarakosol, P., Tantasanawong, P., Han S., A Trustable Software with A Dynamic Loop Control Mechanism, Proceedings of The 5th International Conference on Future Information Technology (FutureTech2010), Busan , KOREA.

2) Srirajun, N., Bhattarakosol, P., Tantasanawong, P., Han S., Trustable Web Services with Dynamic Confidence Time Interval, Proceedings of The 4th International Conference on New Trends in Information Science and Service Science(NISS2010), Gyeongju, KOREA.

3) Srirajun, N., Bhattarakosol, P., Tantasanawong, P., Han S., A Trustable Software with A Dynamic Loop Control Mechanism, International Journal of Information Technology Communications and Convergence, Vol.2, No.1, 2012, pp.54 - 70.

4) Srirajun, N., Bhattarakosol, P., Tantasanawong, P., Han S., Trustable Web Services with Dynamic Confidence Time Interval, Advances in Information Sciences and Service Sciences, Vol 3(4), Advanced Institute of Convergence Information Technology (AIOIT), Korea, 2011, pp. 48-58.

5) Srirajun, N., Bhattarakosol, P., Tantasanawong, P., Han S., EEMTI: an Extending Framework for Nested Web Service Verification, Proceedings of The 7th International Conference on Computing and Convergence Technology (ICCCT2012), 3-5 December 2012, Seoul, Korea, pp. 128-133.

6) Srirajun, N., Bhattarakosol, P., Tantasanawong, P., Han S., A Modification of WSDL for Infinite Loop Verification Under Nested Structures, Journal of Communications and Information Sciences (JCIS), Vol. 3, No. 3, pp. 92 - 98, 2013.

VITA

Name: Miss NATTAPATCH SRIRAJUN.

Date of Birth: 29th December, 1977.

Educations:

- Ph.D., Program Computer Science, Department of Mathematics, Faculty of Science, Chulalongkorn University, Thailand, (June 2008 - October 2013).
- M.Sc. Program Computer Science, Faculty of Science, Silpakorn University, Thailand, (November 2002 - April 2006).
- B.Sc. Program Computer Science, Faculty of Science and Technology, Nakhon Pathom Rajabhat University, Thailand, (June 1995 - March 1998).

Scholarships:

- Office of the Higher Education Commission, Thailand. (June 2008 – October 2011).
- THE 90th ANNIVERSARY OF CHULALONGKORN UNIVERSITY FUND (Ratchadaphiseksomphot Endowment Fund), (July 2012 - October 2013).